8-1-2008

Classification algorithms on the cell processor

Mateusz Wyganowski

Classification Algorithms on the Cell Processor

by

Mateusz Wyganowski

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering
Supervised by

Dr. Muhammad Shaaban
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
August 2008

Approved By:

_____
Dr. Muhammad E. Shaaban
Primary Advisor—R.I.T. Dept. of Computer Engineering

_____
Dr. Juan C. Cockburn
Secondary Advisor—R.I.T. Dept. of Computer Engineering

_____
Dr. Roy W. Melton
Secondary Advisor—R.I.T. Dept. of Computer Engineering

Abstract: The rapid advancement in the capacity and reliability of data storage technology has allowed for the retention of virtually limitless quantity and detail of digital information. Massive information databases are becoming more and more widespread among governmental, educational, scientific, and commercial organizations. By segregating this data into carefully defined input (e.g.: images) and output (e.g.: classification labels) sets, a classification algorithm can be used develop an internal expert model of the data by employing a specialized training algorithm. A properly trained classifier is capable of predicting the output for future input data from the same input domain that it was trained on.

Two popular classifiers are Neural Networks and Support Vector Machines. Both, as with most accurate classifiers, require massive computational resources to carry out the training step and can take months to complete when dealing with extremely large data sets. In most cases, utilizing larger training improves the final accuracy of the trained classifier. However, access to the kinds of computational resources required to do so is expensive and out of reach of private or under funded institutions.

The Cell Broadband Engine (CBE), introduced by Sony, Toshiba, and IBM has recently been introduced into the market. The current most inexpensive iteration is available in the Sony Playstation 3 ® computer entertainment system. The CBE is a novel multi-core architecture which features many hardware enhancements designed to accelerate the processing of massive amounts of data. These characteristics and the cheap and widespread availability of this technology make the Cell a prime candidate for the task of training classifiers.

In this work, the feasibility of the Cell processor in the use of training Neural Networks and Support Vector Machines was explored. In the Neural Network family of classifiers, the fully connected Multilayer Perceptron and Convolution Network were implemented. In the Support Vector Machine family, a Working Set technique known as the Gradient Projection-based Decomposition Technique, as well as the Cascade SVM were implemented.

Chapter 1: Introduction

1.1 The Problem Domain

The strong and steady improvement in data storage technology has lead to an abundance of data being stored by virtually every existing organization. E-commerce companies store transaction details on a per-customer basis. Amazon, for example, generates millions of transactions per day. National Security organizations may collect information about suspected and unsuspected individuals such as travelers or internet users. Banks collect information about their customers which includes spending habits, credit dept and transaction histories. Google, which specializes in data storage, collects information about every search term and every link clicked by every user. Most companies log some portion of incoming and outgoing packet data between their internal intranet(s) and the internet.

There is a lot of valuable information within this data – the problem lies in extracting it. It should be possible, for example, for a bank to predict a new customer's likelihood to go bankrupt by studying all previous similar customers in terms of specified attributes. An advanced intrusion detection system may use a sequence of incoming packets along with timing information to detect novel incoming network attacks. A large e-commerce company may use the data as an aid in choosing which advertisements to show based on the current logged in user (or browser cookie received). The problem is how to search this massive data and create a knowledge model so that future examples from the same domain are classified into some predefined set of classes.

1.2 Classifiers

This is one problem in which classification algorithms, or classifiers, excel. These mathematical tools work by taking as input a set of features of an object, situation, or a piece of data and to producing as an output a discreet value which denotes that input's class. In order to do so, an associated training algorithm is used to generate an internal classifier model, or knowledge, using previous input/output pairings from the same source (problem) domain. This process of building an internal prediction model is known as Data Mining or Machine Learning.

Several classifiers have been invented over the course of the last half century, none of which are perfect, and none of them can be said to outperform the rest in all possible circumstances. Also, there is no optimal method for deciding which algorithm should be used for a particular problem or application. At best heuristics are used but even then a set of algorithm-specific training parameters must be chosen, tested for performance, adjusted, retested, and so on until satisfactory classification performance is achieved. Each training run may take a very long time to complete and often requires access to specialized hardware in order to complete within an acceptable time period. It is never

possible to conclude that the chosen parameters have produced an optimal classifier however. The multiple algorithms and variations available, the parameter adjustability, and most importantly the computational intensity for training make it difficult to explore all possibilities within a reasonable amount of time.

The most popular classification algorithms are the Neural Network (of which the Multilayer Perception is most utilized), Support Vector Machines, k-Nearest Neighbours, Gaussian, Naïve Bayes, Decision Tree and RBF classifiers. This work is concerned with the first two (Multilayer Perceptrons and Support Vector Machines).

1.3 The Cell Broadband Engine

The Cell Broadband Engine Architecture (Cell for short) was released in 2005 by Sony, Toshiba, and IBM. The processor consists of one Power Processor Element, and nine Synergistic Processor Elements (six of which are accessible on the Playstation 3® system used in this work) on one chip tied with a high speed ring bus. The architecture features various novelties that suggest exceptional performance. However, in order to achieve this performance, programs for the Cell Processor need to be explicitly written to take advantage of the hardware. Issues of memory latencies, reformatting of data, workload distribution, all the way down to instruction scheduling need to be carefully considered. The Playstation 3® is arguably the best performance-per-dollar system available due to marketing and business reasons.

The goal of this work is to implement and explore the performance of a two novel Support Vector Machine implementations – the Parallel Gradient Projection-Based Decomposition Technique (PGPDT) and Cascade SVM along with the standard Multi-layer Perceptron and a recent Convolution Layer Neural Network architectures on the Cell Broadband Engine Architecture.

1.4 Organization

The document was written with the assumption that it will be read in order and can be logically divided into four main sections.

Chapters 2 and 3 make up the first section and introduce the two classifiers, placing detail into the variations that are relevant to this work. It is recommended that even those readers familiar with MLPs and SVMs read the sections describing the relatively new MLP convolution layers, and the sections about the novel GPDT working set method and cascade SVM solvers.

The next section, consisting of Chapters 4 and 5, changes course and is concerned with the Cell Broadband Engine and the programming strategies for writing high performance applications on the said architecture. The programming strategies were included in a separate chapter due to the significance that they played in this work.

Chapters 6 and 7 make up the third and most technical section. Here, the implementation of the algorithms described in chapters 2 and 3 are discussed in detail. Graphics were

used as much as possible to convey the concepts, especially where pushing, storage, or processing of data is concerned.

Finally, Chapter 8 closes the document with concrete results that include detailed analysis and educated explanations. As a conclusion, arguments are put forth for the applicability and suitability of the Cell Processor in this field as well as hindsight of what could have been done differently.

Chapter 2: Multi Layer Perceptron

2.1 Chapter Introduction

In this chapter, the Multilayer Perceptron – the most popular member of the Artificial Neural Network (ANN) family – is introduced.

The chapter begins by introducing the basic building block of any ANN – the single Neuron model. This less technical section is broken into the most significant developments and improvements leading up to this day. Details covered are kept within the scope of this work.

The following sections increase in technicality and expand on the concepts introduced in the first section. First, the Single-Layer Perceptron is dissected along with its learning methodology. Next, the Multi-Layer Perceptron is examined by expanding on the Single-Layer version.

Next, the Backpropagation method – the heart of the training power of the MLP – is explained both mathematically and intuitively. Here, the pitfalls and shortcomings of the algorithm, as well as the many variations and parameters that attempt to reduce them are further elaborated.

Typical implementations of the algorithm on modern computer architectures, the feasibility of parallelization, and some recent work are discussed in the next section.

Finally, the convolution layer is described. This layer type was designed to outperform a standard fully-connected layer when 2-dimensional images are the source of the training data.

2.2 Background

The Multi Layer Perceptron is a *supervised* learning algorithm – that is it trains from a previously manually classified *training set*. Every instance of the training set is an input-output pair. This is in contrast to *unsupervised* training algorithms which train on data that has not been previously classified and require that the algorithm generates its own data segregation rules (i.e.: clustering algorithms). The MLP is built from multiple layers and implements a Feedforward network architecture – that is, the network has one input layer and one output layer with no loops such as those appearing in Recurrent networks. Each layer contains some number of neurons.

Two types of layers were implemented in this work – the *fully-connected* layer and the *convolution* layer. In the fully-connected layer each neuron on a layer has a separate dedicated weighted connection to each neuron on the previous layer provided that it is not the input layer. A convolution layer consists of a number of two-dimensional feature maps, each paired with a kernel that is used to generate the values of the feature map by performing a kernel pass over all of the previous layer's feature maps. While in the more general case it is possible to select which feature maps to process for each feature map on the given layer, the implementation in this work processes all input feature maps for each one.

The learning algorithm used in the MLP is known as Backpropagation, which is a gradient descent method having the goal of minimizing an error function at the output layer generated by feeding the network with elements from the training set. The optimization variables are the weights in the fully-connected layers and kernel elements in the convolution layers.

2.3 The Neuron Model

The *Artificial Neuron* is the basic building block of the MLP. The theory behind the MLP can be better understood by first obtaining an intuition into the theory behind the single neuron.

The first neuron model was proposed by McCulloch and Pitts in 1943 [2]. This biologically inspired computational model (from hereon referred to as the M-P model) was very basic, capable of functioning only with binary inputs and outputs. Over the years, the original M-P model has received various modifications from the fields of statistics and probability theory which have allowed it to be applied to a broader range of problems. The modern revision of the neuron unit is shown in Fig. 2-1.

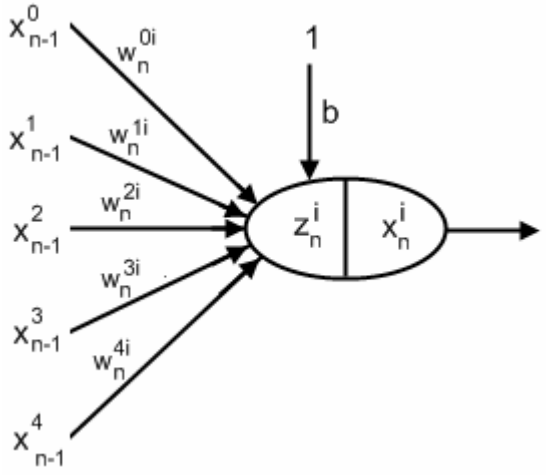


Figure 2-1: The Artificial Neuron Model

The neuron in the figure is the i^{th} neuron on layer $n > 0$ (n=0 represents the input layer). The weighted arrows represent the connections between this neuron and each of the neurons on the previous layer (assuming a fully-connected architecture). An additional

arrow is used to represent the bias *b,* the purpose of which will be discussed later. There are two steps that are taken to calculate the scalar output *x*. The first step involves calculating the activation value *z* as a function of the input vector and weight vector. Most commonly, *z* is calculated by taking a weighted sum of all the inputs (a dot product of the weight and input vector). The result is input to an activation function *F(.)* to produce the final output *x*. For the neuron in the figure, the two steps are summarized by:

$$x_n^j = F\left(\sum_{l=0}^{L_{n-1}}\left(w_n^{l,j} * x_{n-1}^l\right)+b\right) \tag{2.1}$$

in which L_{n-1} is the number of neurons in the previous layer.

The M-P model used only *+1* or *-1* for the weights (known as *excitatory* and *inhibitory* inputs respectively), and used a binary threshold function for an activation. The modern updated model can be thought as a generalization of the M-P model with the ability to work with real numbers. The binary threshold function has been generalized into the set of scalar input, scalar output functions. Typical choices of functions are nonlinear, continuous, and differentiable with an output ranging between -1 and +1. These function characteristics make it possible to apply the *Backpropagation* algorithm, as will be shown in later sections of this chapter.

2.4 The Linear Separator

The neuron model described is nothing more than a biologically-inspired graphical representation of a simple, but powerful, mathematical function. This function, when set equal to 0, represents a hyperplane in an *n*-dimensional space and defines what is called a *linear separator*. The dimension in the case of the artificial neuron is the number of inputs plus one. For example, the equation for a neuron with two inputs is shown in equation 2.2.

$$w_1 x_1 + w_2 x_2 + b = 0 \tag{2.2}$$

Given known weight and bias values, the graph of this function becomes a line (a two-dimensional hyperplane) with the normal **w** and offset *-b/*||*w*|| as shown in Fig. 2.2.

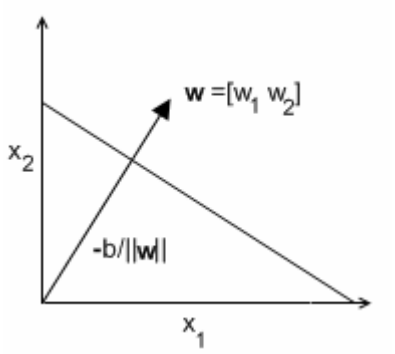


Figure 2-2: Linear Separator of a two-input neuron

Since multiplying both sides of the equation by a nonzero value does not modify the orientation of the line, both sides are divided by the magnitude of the vector **w**, normalizing the weight vector to obtain the same graph, but with updated labels. Fig. 2-3 shows the updated graph along with a new vector (dashed arrow) that is to be classified.

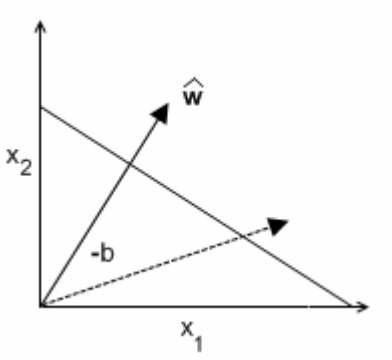


Figure 2-3: Linear Separator after normalizing the weight vector. The dotted line is an input vector that is to be classified.

The power of the linear separator lies in its ability to classify any given input *x* by determining on which side of the hyperplane it sits. In the simple two-input example (Fig. 2-4) it can be shown that given any 2-dimensional input vector the output of the equation $\hat{\mathbf{w}}x - b$ is the perpendicular distance of the point from the line of the linear separator.

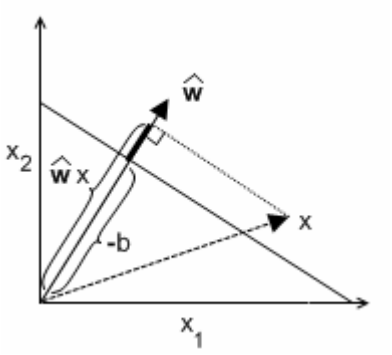


Figure 2-4: Classification of an input vector x.

The dot product between the input and weight vectors gives the magnitude of the projection of the input vector onto the weight normal. Subtracting *b* (adding -*b*) results in a positive value if the vector projects past the line, and vice versa. By encapsulating the function by a sign function, any input can be classified as belonging to either the positive or the negative class. This two-dimensional example generalizes to any number of dimensions making it a powerful tool in various machine learning algorithms.

After defining the outputs of the examples in the training set to be either +1 or -1, a value known as the *margin* can be calculated by taking the product of the signed distance of the input vector, as described in the previous paragraph, and the known output for that vector. The result is positive if the input is correctly classified, negative if not. The

usefulness of the margin value will become clear in the following section in which learning methods are introduced.

2.5 Learning Algorithms

2.5.1 Single Layer Perceptron

The early artificial neuron was very capable but required that the weights were set manually. Many simple systems, such as the digital *AND*, *OR*, and *NOT* gates were easily implemented. However, larger systems required quite a bit more work.

In 1957, Frank Rosenblatt, in his published book, "Principles of neurodynamics: Perceptrons and the theory of brain mechanisms" [3], introduced the *Perceptron* – a model based on the artificial neuron and including a learning algorithm which was the first big step in supervised training methods.

The algorithm for training a single neuron, in a slightly altered form, is shown in Listing 1.

```
Pick initial weight vector (including b), e.g.: [0 … 0]
Repeat until all points correctly classified
Repeat for each Input
                Calculate margin  y_i ⟨w, x⟩_i  for point i
                If margin > 0, point is correctly classified
                Else
change weights to increase margin;
change in weight proportional to  y_i x_i
        Loop
Loop
```

Listing 1: Frank Rosenblatt's Perceptron Learning Algorithm

This is an error-driven algorithm which modifies the weights in an effort to minimize misclassifications. The algorithm was proven to converge to a valid solution after a limited amount of iterations under the condition that the data are linearly separable (there exists a hyperplane that can fully divide the positive and negative set). In the case where the data is not linearly separable, the algorithm never terminates.

The choice of $y_i\mathbf{x}_i$ as the weight increment is attributed to the *gradient ascent* function optimization strategy. The gradient ascent method uses the derivative (gradient) of the function surface with respect to the optimization variable as a means to maximize the function output. Once the gradient of the function is calculated, a step is taken in that direction. The size of the step depends on the gradient slope. In this case the adjustment is made over many input-output pairs, and thus a small fraction of the gradient is actually taken repeatedly over multiple iterations. The gradient is recalculated at every iteration and the algorithm terminates once it becomes small enough (a relatively flat surface is reached).

Recalling the significance of the margin, negative values represent those inputs which were misclassified. A large magnitude implies a big adjustment. It therefore makes sense to define the optimization problem as maximizing the margins for the misclassified points. Optimally, the sum of the margins becomes zero which implies perfect classification of the training set. The function to be optimized is therefore:

$$f(\mathbf{w}) = \sum_{i \text{ misclassified}} y_i \langle \mathbf{w}, \mathbf{x} \rangle_i . \tag{2.3}$$

The gradient of this function with respect to the weights happens to be:

$$\nabla_{\mathbf{w}} f(\mathbf{w}) = \sum_{i \text{ misclassified}} y_i \mathbf{x}_i . \tag{2.4}$$

Rosenblatt's research coined the term *Single Layer Perceptron* in which there are one or more output neurons on the output layer and a multiple inputs on the input layer. Other networks that Rosenblatt experimented with were the cross-coupled Perceptron in which connections joined units of the same layer, and multilayer back-coupled perceptrons, which had feedback paths from units located near the output. While he did propose a back-propagating error method, no one could come up with a way for it to converge.

The discoveries made by Rosenblatt naturally brought new interest into the area of Neural Networks. Unfortunately, this interest dwindled in 1969 when Minksy and Papert published their book *Perceptrons* [4] in which they revealed their discoveries about the limitations of the applications of the Perceptron model. One elegant example used in their arguments was the basic XOR problem. This two input and one output system is linearly inseparable and cannot be learned using the Perceptron model. It is shown in Fig. 2-5.

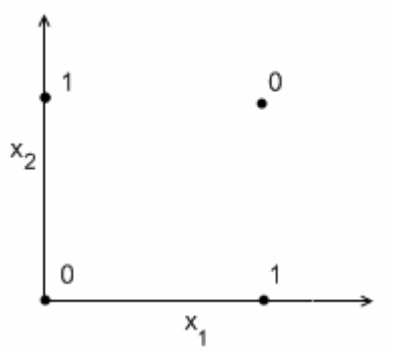


Figure 2-5: Input vectors for the XOR problem

There is no possibility of drawing a straight line so that the four inputs, two of each class, are separated. Applying the Perceptron training algorithm would result in an infinite loop. This simple example, as well as their other discoveries had huge implications for the applicability of the Perceptron to everyday problems. They did propose that adding another *hidden* layer between the input and output layers would theoretically allow for training of linearly inseparable problems, but they had no proposed method for training weights on this layer. An example of a manually configured network capable of dividing

the XOR problem space is shown in Fig. 2-6. The addition of the hidden layer is equivalent to creating linear separators of linear separators. In other words, the output of two decision boundaries (at the hidden layer), is used as input to the final linear separator.

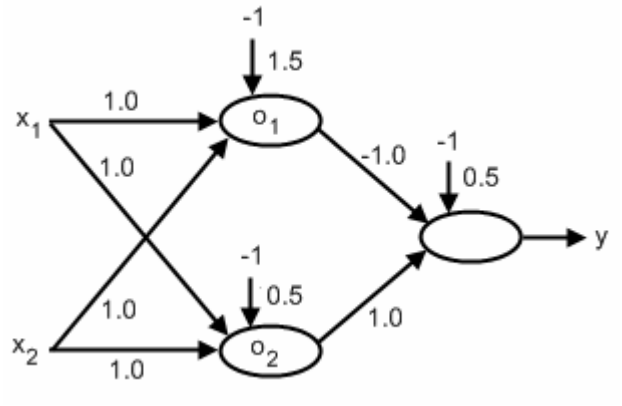


Figure 2-6: A Multilayer solution to the XOR problem

The resulting outputs for each of the possible inputs are:

x_1	x_2	o_1	o_2	y
0	*0*	*0*	*0*	*0*
0	*1*	*0*	*1*	*1*
1	*0*	*0*	*1*	*1*
1	*1*	*1*	*1*	*0*

Figure 2-7: Outputs of each of the linear separators for each input combination

The hidden layer and its weights define two linear separators (Fig. 2-8a). The outputs of these separators are processed by the linear separator of the output neuron (Fig. 2-8b).

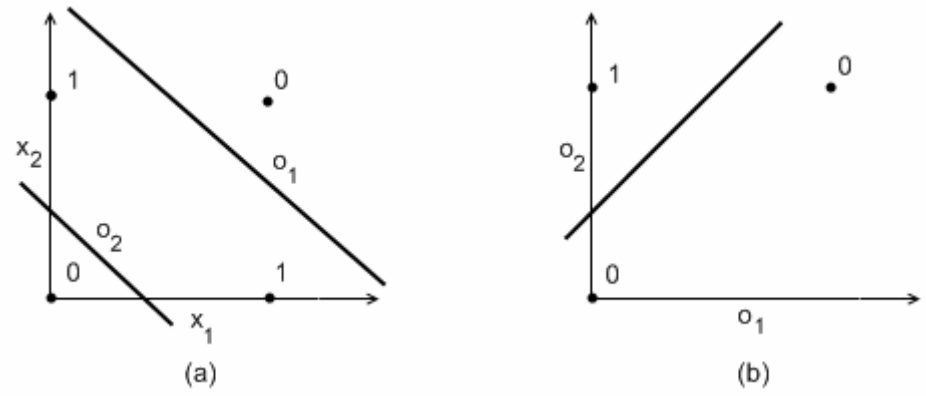


Figure 2-8: (a) Linear separators on the hidden layer. (b) Linear separator at the output neuron.

While their introduction of a hidden layer was an important step, Minksy and Papert's other, rather pessimistic, discoveries were enough to cause widespread decline of further research into the subject.

2.5.2 The Multilayer Perceptron

In 1974, Paul Werbos presented a method for weight adjustment in the hidden layers during a dissertation at Harvard University [5]. Unfortunately, his work went largely unnoticed. It wasn't until 1986 when Rumelhart, Hinton, and Williams published "Learning Internal Representations by Error Propagation" [6] in which they proposed a modified Multilayer Neural Network along with a training algorithm for adjusting weights connected into the hidden layers. Their work was independent to that of Werbos, but they are often credited with the discovery.

The biggest addition in Rumelhart's, Hinton's, and Williams' model is the incorporation of a differentiable activation (or transfer) function. This simple modification allowed them to use an existing mathematical tool called the generalized delta rule (also known as the *heavy ball method* in literature [7] [8] [9]) which is related to gradient ascent. The technique was termed *Backpropagation* in the context of the neural network and remains the most popular method for weight adaptation in the hidden layers. It functions similarly to Rosenblatt's method, but it minimizes the misclassification error rather than maximizing the margin of misclassified training pairs. The ability to adjust weights in the hidden layers opened up new possibilities such as the training of linearly inseparable data. With the birth of the *Multi Layer Perceptron,* interest in the field of neural networks rekindled.

The key to the Backpropagation algorithm is the differentiability of the activation function which implies smooth changes to the output values due to a small weight changes in the network. The optimization function at the output is defined to be the training error over all training vectors:

$$E = \frac{1}{2}\sum_{i=0}^{T}\left(y(\mathbf{x}_i, \mathbf{w}) - d_i\right)^2 \tag{2.5}$$

in which $y(\mathbf{x}_i, \mathbf{w})$ is the network output given the i^{th} input vector, d_i is the expected output, and T is the number of training vectors. It is clear that since $y(\mathbf{x}_i, \mathbf{w})$ is smooth and d_{ii} is constant, the error function is smooth as well. To utilize gradient descent, the gradient of this function is necessary (Eq. 2.6).

$$\nabla_w E = \sum_{i=0}^{T}\left[\left(y(\mathbf{x}_i, \mathbf{w}) - d_i\right)\nabla_w y(\mathbf{x}_i, \mathbf{w})\right] \tag{2.6}$$

The gradient of the error is the sum over all training elements of the differences of the actual and expected outputs multiplied by the gradient of the output with respect to the weights. An expression for the derivative of the output with respect to the weights needs to be found. Recall that an output y is the weighted sum of the inputs processed by the activation function.

The gradient for the output with respect to the weights is obtained by applying the chain rule of differentiation as shown in Eq. 2.7 and Fig. 2-9.

(2.7)
$$\begin{aligned}\frac{\partial y}{\partial w_i} &= \left(\frac{\partial y}{\partial z}\right)\left(\frac{\partial z}{w_i}\right) \\ &= \left(\frac{\partial F(z)}{\partial z}\right)\left(\frac{\partial z}{\partial w_i}\right) \\ &= \left(\frac{\partial F(z)}{\partial z}\right)(x_i)\end{aligned}$$

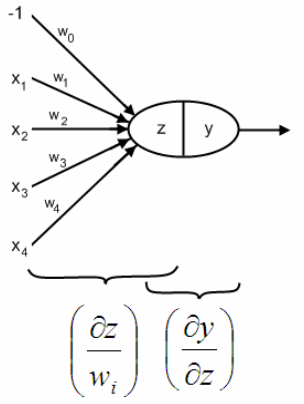


Figure 2-9: Calculating the output gradient with respect to the weights.

The derivative is broken down into a product of the derivative of the output with respect to the activation value (the slope of the activation function with the current value z) and the derivative of the activation value with respect to the current weights (for a Single Layer Perceptron, this value is simply the current input x). By the rule of gradient descent, for every neuron at the output layer, the weight vector should be updated by an element-by-element product of the output error vector and the activation function gradient at that activation value.

(2.8)
$$\begin{aligned}&\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_w E \\ &w_i \leftarrow w_i - \eta(y-d)\frac{\partial F(z)}{\partial z} x_i\end{aligned}$$

The parameter η is a small value known as the *learning rate*. If a hidden layer is introduced, which includes its own set of weights, a method is required for obtaining the gradient of the output y with respect to one of the hidden layer weights. The same procedure is used, but since the input into the output layer is an output from the hidden, a recursive loop is observed as shown in Eq. 2.9 and accompanying Fig. 2-10.

(2.9)

$$\frac{\partial y}{\partial w_{ij}} = \left(\frac{\partial y}{\partial z}\right)\left(\frac{\partial z}{w_{ij}}\right)$$
$$= \left(\frac{\partial F(z)}{\partial z}\right)\left(\frac{\partial z}{\partial w_{ij}}\right)$$
$$= \left(\frac{\partial F(z)}{\partial z}\right)\left(w_i \frac{\partial y_i}{\partial w_{ij}}\right)$$
$$\rightarrow \left(\frac{\partial F(z)}{\partial z} w_i\right)\left(\frac{\partial y_i}{\partial w_{ij}}\right)$$

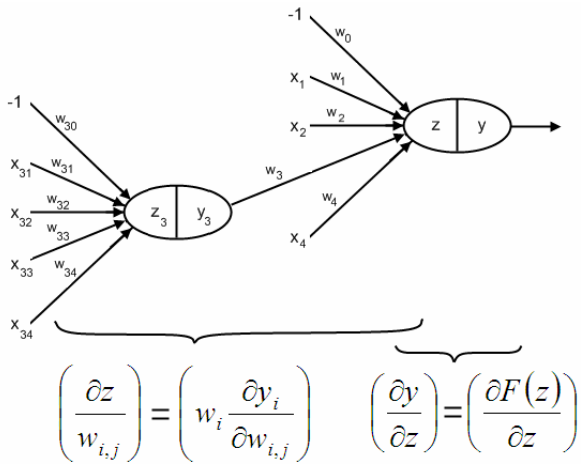


Figure 2-10: Chain rule for the output gradient with respect to the weights.

In more general networks, multiple weights near the input may affect the signal coming into a neuron on a further layer. In such cases, the gradient of the signal can be obtained by summing the effects over all possible paths.

The intuition is built on the observation that the output gradient is a function of the signal on each of the inputs into the output neuron. Changing a weight on a hidden layer of some hidden neuron produces a gradient on the signal between the hidden neuron and the output neuron. Assuming that there is only one hidden layer as in most networks, the gradient on any of the signals is calculated using the base case above.

In order to implement Backpropagation a new quantity δ is introduced. At the output layer, the delta for a single neuron is:

(2.10) $$\delta = \frac{\partial E}{\partial z} = (y - d)\frac{\partial F(z)}{\partial z}.$$

To obtain the deltas for the j^{th} neuron in the hidden layer l, the deltas from the $(l+1)^{th}$ layer are propagated backward and multiplied by the derivative of the activation function acting on the input of the neuron:

(2.11) $$\delta_j^l = \frac{\partial F(z_j^l)}{\partial z_j^l}\sum_{k=0}^{L_{l+1}} \delta_k^{l+1} w_{jk}.$$

By substituting Eq. 2.10 into the general case of Eq. 2.8, the weight connecting neuron i from layer l to neuron j in layer $l+1$ is updated by:

(2.12) $$w_{ij} \leftarrow w_{ij} - \eta \delta_j y_i.$$

The name Backpropagation comes from the way in which the δ value is propagated backwards through the network. Reusing the deltas as they are propagated back is the grounds for its efficiency. Listing 2 summarizes the Backpropagation algorithm.

```
Initialize weights to small random values
Repeat until error over entire training set is small enough
        Repeat for all Training input-output pairs
                Propagate forward by computing the outputs of each layer in succession
                Compute δ at the output layer
                Propagate the δ value backward through the layers
        Updating the weights on each layer
        Loop
Loop
```

Listing 2: Backpropagation Summary

2.6 Training the Multi Layer Perceptron

This section summarizes some of the issues that plague MLP training as well as some of the parameters and modifications to the training algorithm that the implementer has control over. One of the major drawbacks of training the MLP is the lack of automated training parameter discovery. The implementer is forced to experiment with such things as numbers of layers, layer sizes, learning rates, methods of learning, convergence conditions, length of training, etc. The decisions made are mostly based on heuristics and experience. Options exist that may simplify this process, but usually involve tradeoffs and never completely eradicate the problem.

Several issues due to improper setup as well as those inherent in the MLP are presented, along with methods that aim to reduce them. As will become evident, many design choices are influenced by what is desired and how much time and resources are available.

2.6.1 Combination Functions

The combination function defines the transformation of the two vectors ($\mathbf{w}$,$\mathbf{x}$) to scalar (z) in the first neuron processing step. The MLP, in nearly all cases, uses the dot-product. However, other networks exist – most popular being the Radial Basis Function network in which, as the name implies, the RBF function is used instead [10].

2.6.2 Activation Functions

As discussed in the previous section, the activation function is utilized in the second step of the neuron's forward process and its derivative is utilized during Backpropagation. The activation function in the hidden layer needs to be nonlinear for the network to be capable of learning from datasets that may not be linearly separable. Activation functions that produce values centered around zero (*i.e.*: [-1.0,1.0]) are usually preferred as they generally converge quicker due to improved numerical conditioning [11]. The two most popular activation functions are the sigmoid and hyperbolic tangent. Table 1 shows these two functions as well as an approximated version (used in this work) [12].

	$F(z)$	$\frac{\partial F(z)}{\partial z}$
Sigmoid	$\frac{1}{1+e^{-z}}$	$F(z)(1-F(z))$
Hyperbolic Tangent	$\tanh(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$1-\tanh^{2}(z)$ $=1-F(z)^{2}$
Hyperbolic Tangent (LeCun's Appx..)	$=1.7159\tanh\left(\frac{2}{3}z\right)$	$\left(\frac{2}{3}\right)\left(\frac{1}{1.7159}\right)(1.7159+F(z))(1.7159-F(z))$

Table 1: Activation Functions

The popularity of these activation functions is attributed to the ease of obtaining the derivative using values already in memory. The graph of each is shown in Fig. 2.11.

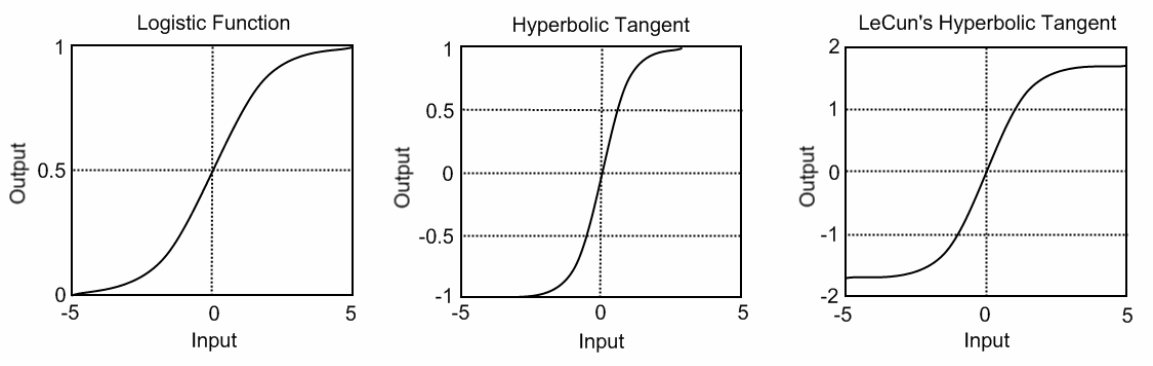


Figure 2-11: Activation function curves

2.6.3 Bias Term

As described in the previous section, each neuron in the MLP model contains a *bias* term that shifts the linear separator's hyperplane along the weight normal. In many implementations of neural networks, the bias is represented simply as another weighted input with the constant value of +1 or -1 (neither have an advantage when training). The corresponding weight is trained just like any other on that neuron. The bias term is optional, although the theory of the universal approximation does not hold without it [13]. One way in which the bias may be avoided is by forcing the condition that the outputs on any layer sum to a nonzero constant [14]. This is most easily done by preprocessing the input data so that each vector has elements which sum to the same constant value.

The neuron processing equation, with the bias term treated as a weighted input is

$$x_n^j = F\left(\sum_{l=0}^{L_{n-1}+1}\left(w_n^{l,j} * x_{n-1}^l\right)\right) \tag{2.13}$$

in which $w_n^{L_{n-1}+1,j}$ is the weight associated with the bias term. This simplified equation will make the implementation much simpler as will be seen in Chapter 6.

2.6.4 Initial Weights

The choice for initial weight values is quite important when initializing the network since it decides the initial position in the search space. Values should be small enough so to prevent the neuron outputs from saturating. Recall that the slope of the activation function becomes flat at the input extremes. Because weight updates are proportional to this slope, learning may become extremely sluggish. A good choice of weights, therefore, is one that produces midrange function signals. A good practice is to select zero centered random weights within a small range.

2.6.5 Local Minima

One of the major issues that plague MLP training is the possibility of the gradient descent method to become trapped in a *local minimum*. This is a direct result of the fact that the error surface that is being searched is non-convex. The strategies described in the following sections may help in this matter.

2.6.5.1 Learning Rate

An important parameter that needs to be selected if performing incremental learning (in which weights are updated after every training pair) is the *learning rate* η. The purpose of this parameter is to subdue the effect of the gradient descent on the values of the weights. Every application of the gradient descent method is valid only locally for the input-output pair currently applied. Too big of a change would likely negatively affect the accuracy on the remaining set. Values are typically in the range of (0, 1].

The learning rate value should be selected based on, among other things, the number and size of the layers and the number of training vectors. Too high of a value can cause chaotic oscillations around the solution. Values too low may cause a very long training time.

One option is to implement a dynamic learning rate that uses information about the conversion progress to adapt the learning rate. There is a good amount of research on the topic. However, a simple and effective method is to decrease the learning rate periodically over time. This practice also guarantees convergence.

2.6.5.2 Momentum Term

By using nothing but the current error gradient information in choosing the direction and step size on the error surface, the weight vector can become very erratic and constantly change direction at the application of each new training sample. Having no “mass,” the search has no memory of the tendency of the search direction and can be thought of as taking the first opportunity of a minimum that it gets. In order to introduce memory, the weight step at each sample is stored for use by the next iteration. The *momentum* term specifies how much influence the previous step has in the current iteration’s step vector. Having a momentum has the effect of averaging out the gradients through time helping

the search overcome any minimums encountered along the general downward slope of the gradient. The updated weight updating step, utilizing the deltas of the previous weight updates is:

$$\begin{aligned} \Delta \mathbf{w}_t &= -\eta \nabla_w E_i + \alpha \Delta \mathbf{w}_{t-1} \\ \mathbf{w}_{t+1} &\leftarrow \mathbf{w}_t + \Delta \mathbf{w}_t \end{aligned} . \tag{2.14}$$

Note the addition of the momentum term α which needs to be selected. The effect of the change is that sequential weight changes in the same direction accelerate, while those of opposite signs cause the changes to decelerate, as expected.

2.6.5.3 Multiple Training Sessions

Another, simple and brute force, method to improve the chance of finding the global minimum is to train on the same data modifying only the initial weight values. Of course, this method can significantly increase the learning time and has no guarantees.

2.6.6 Incremental vs. Batch Learning

Weight updates may be performed after the calculation of the error of each individual input-output sample or at the end of a full *epoch*. The former case is called *incremental* learning (also instantaneous or pattern learning). In this mode, the delta is propagated back as described in the previous section, and the weights are updated following the Backpropagation of each training vector's output error. It is important that the deltas are "pushed" onto the neurons of the previous layer before modifying the weights themselves on any particular layer so that the error "blame" is placed on the neurons for which the connections were strongest during the forward propagation.

When performing incremental training, a recommended strategy is to present the network with the training samples in random order within the epoch. This prevents the network from giving the early samples in the set a greater priority and may reduce the risk of getting stuck in a local minimum.

In batch learning mode (also epoch learning), the samples from the entire training set are propagated forward and back through the network accumulating all the weight adjustments for each layer. Training samples do not need to be applied randomly as this would make no difference in the final values. The asynchronous nature between iterations of one epoch makes batch learning a better candidate for parallel implementations. The greatest benefit to batch learning, however, is the possibility of applying adaptive learning methods that are described next.

2.6.6.1 Adaptive Learning Techniques

By incorporating batch learning, several newer *adaptive* weight adjustment rules can be applied that have shown to provide quicker conversion. While theoretically not as accurate as incremental learning (if good incremental learning parameters are chosen), they can be orders of magnitude faster at reaching convergence. Two techniques that receive much praise are the Resilient Backpropagation (RPROP) [15] and Quickprop (Qprop) [16] algorithms.

Standard BP uses the derivative of the activation function as a part of the weight updating calculation. Due to the shallow slope at high magnitude input values, the change in weight may be very small for large error values. RPROP, on the other hand, uses only the sign of the *dE/dw* value along with an externally defined value to update the weights. The weight change is calculated as shown in Eq. 2.15.

$$w_{ij}^{t} = \begin{cases} -\Delta_{ij}^{t}, & \text{if } \dfrac{\partial E}{\partial w_{ij}} > 0 \\ +\Delta_{ij}^{t}, & \text{if } \dfrac{\partial E}{\partial w_{ij}} < 0 \\ 0, & \text{else} \end{cases} \tag{2.15}$$

The adaptive feature of the RPROP algorithm comes from the self-adjustment of the weight delta values as follows:

$$\Delta_{ij}^{t} = \begin{cases} \eta^{+}\Delta_{ij}^{t-1}, & \text{if } \left(\dfrac{\partial E}{\partial w_{ij}}^{t-1}\right)\left(\dfrac{\partial E}{\partial w_{ij}}^{t}\right) > 0 \\ \eta^{-}\Delta_{ij}^{t-1}, & \text{if } \left(\dfrac{\partial E}{\partial w_{ij}}^{t-1}\right)\left(\dfrac{\partial E}{\partial w_{ij}}^{t}\right) < 0 \\ \Delta_{ij}^{t-1}, & \text{else} \end{cases} \tag{2.16}$$

in which $0 < \eta^{-} < 1 < \eta^{+}$. In words, if the slope of the error curve with respect to some weight negated, this signifies an overshoot and the multiplier for that weight is decreased. In this situation, the weight is not modified at this epoch. If the slope direction remained the same, the conversion can be accelerated by increasing the multiplier for that weight. Some parameters that can be modified are the values of η^{+} and η^{-}, and the maximum and minimum of the deltas.

The Quickprop (or Qprop) algorithm is a secant method which uses the property that the gradient of the error surface with respect to the weights is zero at all local minima. One way to obtain the optimal weight values is to solve the following system of equations:

$$\begin{aligned} &\partial_1 E(w_0, w_1, ..., w_n) = 0 \\ &\partial_2 E(w_0, w_1, ..., w_n) = 0 \\ &... \\ &\partial_n E(w_0, w_1, ..., w_n) = 0 \end{aligned} \tag{2.17}$$

in which $\partial_i E$ is the i^{th} coordinate of the error gradient. The weights are updated as follows:

$$w_i^{t+1} = w_i^t - \eta_i \left\{ \frac{\partial_i E(w^t) - \partial_i E(w^{t-1})}{w_i^t - w_i^{t-1}} \right\}^{-1} \partial_i E(w^t). \tag{2.18}$$

2.6.7 Generalization and Overfitting

2.6.7.1 Definition

The typical purpose of a neural network is to be accurate not only on the training set, but any new novel input from the same population. A network capable of classifying many novel inputs correctly has a good *generalizing* characteristic. Training a network so that it retains high generalizing capability is not easy. There is much research in this area that is beyond the scope of this thesis. The following are just several recommendations that should be followed when choosing the training data [14]:

1. The choice for attributes in the input data must make sense with that of the output class. In other words, there needs to be some correlation, even if theoretical, between the cause and effect.

2. For the case of continuous functions, the function that is being learned should be smooth. While this is not a necessity, it is beneficial for the Backpropagation algorithm which relies on the gradient of the function as a choice for weight updates. In some cases, a preprocessing of the input data, such as by using Principal Component Analysis, can help.

3. The training set should be sufficiently large and be a good representation of the data that the neural network is expected to come across. A good representation is defined as input samples that cover a large part of the input space and are evenly separated.

Following these rules requires some prior knowledge of the function being approximated.

Generalization can be lost if the training algorithm is run for too long on the training data. In this phenomenon, known as *overfitting*, the surface of the hypothesis function, in an effort to "touch" every point in the training set, may become jagged having many irregularities between the training points. This in turn leads to the loss of interpolating ability and lack of generalization. It is therefore necessary to prematurely stop the training process before it enters the overfitting stage. This is difficult, if not impossible, to achieve if using only the errors in the training set as the criteria for the stopping condition.

2.6.7.2 Cross-Validation and Early Stopping

A common and simple method used to reduce overfitting is *cross-validation* – a technique first introduced by Seymour Geisser [17] used in many machine learning algorithms. The technique involves segmenting the original training data into training and validations sets. The idea is to train on the training set and monitor the generalization on the validation set. The information gained from the validation set can be used to select

model parameters (such as network size), or for determining when to stop training (*early stopping*).

Various partitioning schemes exist based on the cross-validation technique. In the simplest method, called Holdout Validation, the initial training data is split into the two aforementioned sets. The training data is used to train the model. After each iteration of the training set the validation set is used to compute the accuracy. Fig. 2-11 shows typical behavior of the errors on the training and validation sets as a function of the iteration number. Early stopping often incorporates the holdout validation scheme. The point at which the algorithm should stop is marked by a vertical dotted line. Note that the error on the training set continues to decrease.

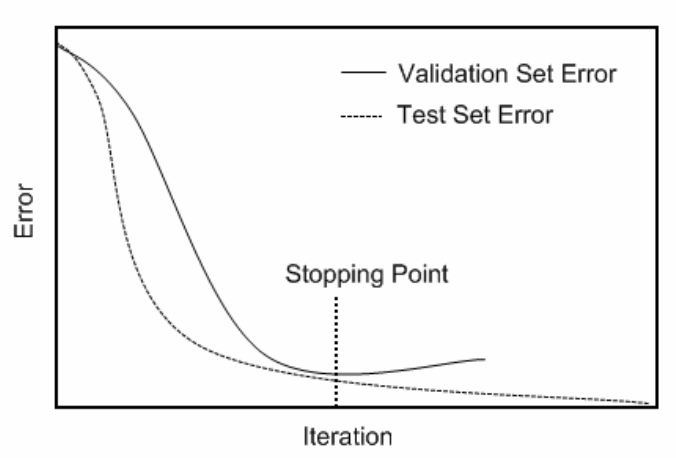


Figure 2-12: Training Set error and Validation Set error as a function of training iteration number.

A more advanced partitioning scheme, known as K-Fold cross-validation, entails partitioning the original set into *k* subsets. The same algorithm is run *k* times by leaving one of the *k* subsets out and using it as the validation set. The results from the *k* runs are accumulated and processed (often by taking an average). While it is not obvious how to use this partitioning scheme for early stopping, it can be used to evaluate the network structure for generalization.

A third partitioning scheme, known as Leave-one-out cross-validation, is simply K-Fold cross-validation with *k* set to the number of training samples. In other words, the network is trained *k* times, reserving only one of the training samples for validation. As expected, this method takes a very long time and is more applicable to smaller problems.

2.6.8 Jitter

This simple, but powerful method works on the principle that input vectors that are close together should produce outputs that are close together. The method involves modifying the input vectors by changing their continuous attribute values in very small percentages. This can also be effective when the training set available is quite small and works particularly well with images where the 2-dimensional image is systematically rotated, stretched, etc.

2.6.9 Weight Decay

Large weights in an MLP are known to negatively affect generalization performance. Having large weights at the hidden layer can easily cause the hypothesis function to become rough with small changes in the input and have near discontinuities. Large weights leading into the output layer can cause the output to become too large and leave the range of the possible outputs. *Weight Decay* is a method in which the error function is augmented with a weight penalty term. An example of a penalty term is a fraction of the squared sum of the weights (Eq. 2.19). This simple addition forces the network to try and keep down the weight values and prevents these kinds of problems.

$$E = \frac{1}{2}\sum_{i=0}^{T}\left(y(\mathbf{x}_i, \mathbf{w}) - d_i\right)^2 + \sum_{k} w_k \tag{2.19}$$

in which k is the set of indices for all the weight parameters.

2.7 Network Architecture

The most obvious and arguably most critical parameter that the implementer has control over is the number of neurons in the hidden layer, or whether to include a hidden layer at all. Naturally, choosing to forego the hidden layer would speed up the learning process considerably. In contrast, there are rare cases in which there is a need for multiple hidden layers. The size of the input layer is set by the dimension of the input data as is the output layer due to the output data. The hidden layer, however, is under full control. Selecting a proper number of units in the hidden layer(s) takes into consideration the dimension of the inputs and outputs, size of the training set, complexity of the function to be learned, number of layers and network architecture, type of activation function used, training algorithm, and other factors. Often, the best action is to simply train multiple networks and see which one works best.

As part of an effort to automate the task of choosing the network architecture, new mechanisms which dynamically adapt the number of neurons as well as connections between them have been developed. These generally fall into constructive and pruning algorithms in which neurons are added or removed automatically as the learning progresses. These techniques are beyond the scope of this work. A summary of popular constructive algorithms is presented in [18], [19] and [20]. Pruning algorithms are summarized in [21] and [20].

These are only some of the options that the designer is able to tune and experiment with. There is much research being done in many areas, whether for speed or accuracy of training or simplifying the implementation steps by making the network more adaptive to the problem at hand. When tweaking a network that is to function on large data sets (or having a large input space), considering that a single training session can take days or weeks to complete, experimentation can be very time consuming and would benefit largely from an optimized modular and expendable toolset running on affordable hardware.

2.8 Convolutional Networks

The standard Multi-Layer Perceptron model consists of two or more layers of neurons. Each layer is usually fully-connected to the layer before it, meaning that every neuron on layer *l* is connected to every neuron on layer *l-1*.

Traditionally, when applying neural networks to 2-d inputs, such as images, so called feature extractors are placed between the raw image input and the input layer of the network. These feature extractors, or kernels, are hand tuned to extract information such as edges and discard irrelevant variables. The process is called convolution.

LeCun et al. [22] [23] have tried to remove this extra step of needing to manually create the kernels. Instead, their design, known as the Convolutional network, uses Backpropagation to automatically train the feature extractors.

Convolution networks (consisting of convolution and subsampling layers) are similar to fully-connected networks (consisting of fully-connected layers), and can in fact be interconnected with convolution layers appearing before fully-connected layers in the network. Convolutional layers borrow the idea of an image feature extractor (or kernel) from image processing and train these kernels automatically (just as fully-connected layers train the weights going into them).

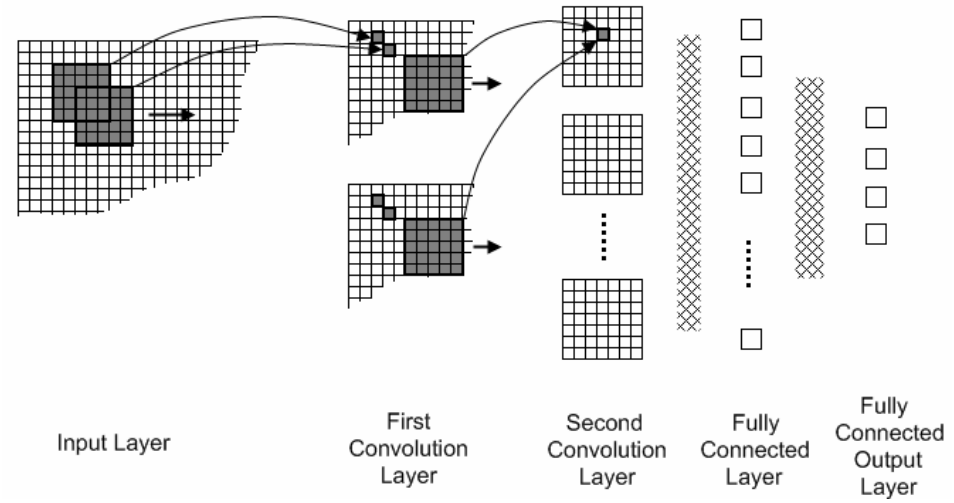


Figure 2-13: Convolution Layer network architecture.

Fig. 2-12 shows the concept of convolution layers. In the architecture, each layer has multiple kernels that it trains independently. Each kernel is paired with its own feature map which is the output after the kernel traverses over an input image. The feature maps are analogous to neuron values in the fully-connected network. The kernels are analogous to the weights although there are a significantly smaller number of them being that the same *n*x*n* kernel is applied to every possible block of *n*x*n* input pixels. By substantially reducing the number of weights, the network is much less prone to Overfitting, is quicker to learn, and greatly decreases memory requirements.

The main advantage to Convolutional layers in comparison to standard layers, however, is their inherent invariance to image transformations such as shifts, scales, and rotates. A standard layer, upon learning from a training set of shapes, for example, would result in having similar weights repeated at certain intervals. On the other hand, the Convolutional layer kernel which is made of a relatively small set of weights traverses the entire image. This shared weight concept is not new, but is found to work well for 2-dimensional inputs. An example of LeCun's networks is shown in Fig. 2-13. It is the LeNet-5 [12] which was designed to recognize handwritten characters.

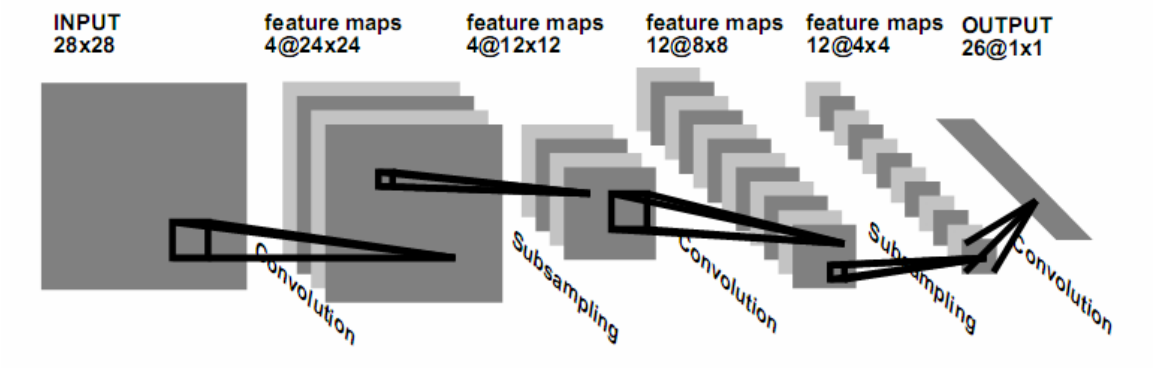


Figure 2-14: LeNet-5 Network

Note that the LeNet-5 architecture consists of convolution layers, which function as described above, and subsampling layers, which take averages of closely spaced pixels to produce a smaller feature map. Subsampling helps reduce the effect of shifts of the input image between training samples. The implementation in this work achieves a similar effect by combining the convolution and subsampling layers into one by moving the *n*x*n* kernel two pixels at a time in the *x* and *y* directions.

2.9 Related Work

Being a relatively easily parallelizable algorithm, the standard MLP algorithm (consisting of only fully-connected layers) has been parallelized on a vast number of devices, including graphics processing units [24], computer clusters [25], distributed systems [26], FPGAs [27], etc. Convolutional network implementation, however, remain scarce largely due to the increased complexity involved for Backpropagation.

Chapter 3: Support Vector Machines

3.1 Chapter Introduction

Support Vector Machines (SVMs) are similar to the Multi-Layer Perceptron (MLP) in that they generalize into the family of linear classifiers, or separators, introduced in the previous chapter. The methodology for training, however, is quite different. While MLPs are largely based on heuristics (even the very first neuron model was based on heuristics), SVMs borrow proven theorems and tools from the fields of optimization theory, generalization theory, and statistical learning theory making them more understood. SVMs, as will be revealed, hold many important advantages compared to other training methods such as the MLP. The chapter discusses early motivation for this learning system and introduces some of the concepts which serve as the foundations for its functionality.

3.2 Background

It is difficult to pinpoint the exact point in history at which Support Vector Machines appeared, but much of its advancement is credited to Vladimir Vapnik who is considered by many to be the main inventor and contributor. It has been widely accepted that the beginnings of SVMs appeared with the publishing of the book "Estimation of Dependencies Based on Empirical Data" (1979) [28] in which Vapnik provided the foundations of statistical learning theory. The SVM algorithm itself was introduced in [29]. The main purpose of the SVM was to overcome some of the shortcomings of existing classifiers – especially those of the MLP. Both the SVM and the MLP belong in the class of linear learning machines that were introduced in the previous chapter but are very different in the techniques that they employ. This difference is largely attributed to the dissimilar processes by which the two came about. SVMs were designed using existing proven theories and concepts taken from optimization theory, generalization theory, statistical learning theory, etc. The MLP takes concepts from existing theories as well, but it is also largely based on heuristic models such as the very first McCulloch and Pitts neuron model itself (which in some way was based on the yet little understood neuron inside the brain).

3.3 Training

From a high level perspective, training SVMs is very similar to training the MLP. A set of training samples is provided along with a number of training parameters, the learning process is started, and after some time a classifier model that can be used for future classification is obtained. It is the internal part, and the representation of the final trained

model that really set the two apart. Also, unlike the MLP, the SVM is inherently only capable of distinguishing between two classes, making it a *binary classifier*. For the purpose of this chapter, elements are either classified as either *positive (+)* or *negative (-)*.

Like the MLP, the goal of SVM training is to produce a linear n-1 dimensional hyperplane in an n-dimensional input space that separates the two classes. The SVM is known as a *maximal-margin classifier* because it attempts to find a hyperplane that not only classifies each training vector correctly, but it optimizes its position and orientation by taking into account the perpendicular distances of the closest points to the hyperplane. In other words, the algorithm attempts to find the widest possible "street" (of widest margin) that can still classify all training vectors. This constraint can be relaxed, as will be explained later. The name "support vectors" comes from the algorithm's objective to discard any vectors that do not affect the margin; those that do not touch the edges of the "street." A toy example of a linearly separable two-dimensional problem is shown in Fig. 3.1 with the support vectors circled in red.

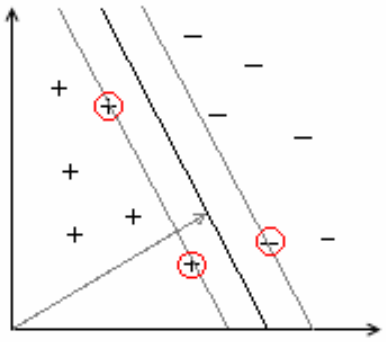


Figure 3-1: A toy example of a widest-margin classifier in two dimensions.

3.3.1 Linearly Separable Case

The SVM can be applied to many real problems. However, for the purpose of this section, the theory is first introduced in the simplest case in which all training vectors are linearly separable. Once the main concept is introduced, further modifications will be described that extend the SVMs application to more realistic cases.

As a review, the hyperplane of a linear learning machine is defined by the function:

$$f(x) = \sum_{k=1}^{d} w_k x_k + b \qquad or \qquad f(x) = \langle \mathbf{x} \cdot \mathbf{w} \rangle + b \tag{3.1}$$

in which d is the number of dimensions of the input space. The perpendicular distance from the hyperplane to the origin is $|b|/\|w\|$ in which $\|w\|$ is the Euclidean norm of the vector w. Given a set of training vectors within the input space that are correctly classified by the hyperplane, let d_+ be the distance to the closest positively classified

training vector and d_- be the distance to the closest negatively classified training vector, maximizing the margin implies maximizing $d_+ + d_-$. For the purposes of optimization, the goal is to find a representation of the margin width in term of the optimization variables **w** and/or *b*. It is a known property of linear classifiers that scaling the values **w**, *b* by the same positive factor does not change the function. On the other hand, the values of d_+ and d_- will be impacted. For derivation purposes, d_+ and d_- can be forced to 1, resulting in the following two inequalities:

$$\begin{aligned} &(a)\ \langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b \geq +1 && \{i \mid y_i = +1\} \\ &(b)\ \langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b \leq -1 && \{i \mid y_i = -1\} \end{aligned} \tag{3.2}$$

which can be combined into one convenient expression of Eq. 3.3.

$$y_i\left(\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b\right) - 1 \geq 0 \quad \forall i \tag{3.3}$$

Assume that there exist points for which the *equality* in Eq. 3.2a holds and another set of points for which the *equality* in Eq. 3.2b holds (this implies a trained **w** and *b*). The points satisfying Eq. 3.2a lie on the hyperplane $\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b = 1$ which has a perpendicular distance from the origin of $|1-b|/\|w\|$. Similarly, those satisfying Eq. 3.2b lie on the hyperplane $\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b = -1$ which has a perpendicular distance from the origin of $|-1-b|/\|w\|$. These points are known as the support vectors as they touch the extremities of the margin. All other points (satisfying the inequality of Eq. 3.2a and 3.2b) are non-support vectors and would not change the final values of **w** and b should they be removed.

Since both hyperplanes are parallel, the distance, *m*, between them is obtained by taking the difference between their perpendicular distances from the origin.

$$\begin{aligned} m &= \left(|1-b|/\|w\|\right) - \left(-|1-b|/\|w\|\right) \\ &= 2/\|w\| \end{aligned} \tag{3.4}$$

Maximizing the hyperplane, therefore, amounts to minimizing $\|w\|^2/2$.

The problem can be reformulated into Lagrangian form as shown in Eq. 3.5. By doing so the constraint in Eq. 3.2 is replaced with constraints of Lagrange multipliers $\alpha_i, i = 1,...,l$ (one for each of the *l* input training vectors) which are easier to handle. Also, the training data will appear during the training and classification phases in the form of dot products – a property that is central to the SVM's ability to generalize to the nonlinear case as will be shown in later sections. The primary Lagrangian is

$$L_P \equiv \frac{1}{2}\|w\|^2 - \sum_{i=1}^{l} \alpha_i y_i \left(\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b\right) + \sum_{i=1}^{l} \alpha_i \tag{3.5}$$

Because the objective function itself is convex and those points which satisfy the constraints are convex, the problem becomes a quadratic programming problem. With the help of optimization theory, the problem can be reformulated once again into a dual form which is known as the Wolfe dual [30]. The dual formulation is often easier to solve due to the difficulty in handling inequality constraints. The procedure for transforming a primal into a dual is to zero the derivatives of the primal function with respect to the optimization variables, and substitute the resulting relations into the primal. This process removes the dependence on these variables. For the primal obtained above, the process is as follows.

Differentiating the primal form with respect to **w** and b and setting it to zero gives:

$$\frac{\partial L_P(\mathbf{w},b,\alpha)}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{l} y_i \alpha_i \mathbf{x}_i = 0 \qquad \Rightarrow \qquad \mathbf{w} = \sum_{i \in l} y_i \alpha_i \mathbf{x}_i$$

$$\frac{\partial L_P(\mathbf{w},b,\alpha)}{\partial b} = \sum_{i=1}^{l} y_i \alpha_i = 0 \qquad . \tag{3.6}$$

Substituting back into the primal results in the dual problem:

$$\begin{aligned} L_D &\equiv \frac{1}{2}(\mathbf{w}\cdot\mathbf{w}) - \sum_{i=1}^{l} \alpha_i \left[y_i \left(\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b \right) - 1 \right] \\ &= \frac{1}{2} \sum_{(i,j)=(1,1)}^{(l,l)} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle + \sum_{i=1}^{l} \alpha_i \\ &= \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{(i,j)=(1,1)}^{(l,l)} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle \end{aligned} \qquad . \tag{3.7}$$

The constraints for this problem, borrowed from the primal version, are:

$$\begin{aligned} &\alpha_i \geq 0, \ i \in l \\ &\sum_{i=1}^{l} \alpha_i y_i = 0 \end{aligned} \qquad . \tag{3.8}$$

The Wolfe dual maximum occurs at the same point of **w**, b and α_i as the primal Lagrangian's minimum subject to the two problems' corresponding constraints. The values of α_i at the solution are greater than zero for those training vectors which lie on one of the two hyperplanes (the support vectors) and equal to zero for those vectors which touch one or are outside of the two hyperplanes (all other vectors which would not influence the resulting hyperplane if removed).

The Karush-Kuhn-Tucker (KKT) conditions, also borrowed from the field of nonlinear programming techniques, are conditions that if satisfied, are necessary and sufficient for optimality. For the case of the primal problem, they are:

$$
\begin{aligned}
&\frac{\partial L_P}{\partial w_v} = w_v - \sum_{i=1}^{l} \alpha_i y_i x_{iv} = 0, \qquad v = 1,...,d \\
&\frac{\partial L_P}{\partial b} = \sum_{i=1}^{l} \alpha_i y_i = 0 \\
&y_i\left(\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b\right) - 1 \geq 0, \quad i = 1,...,l \\
&\alpha_i \geq 0, \qquad\qquad \forall i \\
&\alpha_i\left[y_i\left(\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b\right) - 1\right] = 0, \qquad \forall i
\end{aligned}
\tag{3.9}
$$

Once the α_i 's are found, **w** is obtained directly from one of the primal constraints:

$$
\mathbf{w} = \sum_{i=1}^{l} y_i \alpha_i \mathbf{x}_i \tag{3.10}
$$

Once **w** is calculated, b can be obtained. Since the *b* does not appear in the dual problem, a safe way to obtain it is to use the mean of the solutions of the "complimentarity" KKT condition:

$$
\alpha_i\left[y_i\left(\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b\right) - 1\right] = 0 \tag{3.11}
$$

for *i*'s corresponding to non-zero Lagrange multipliers ($\alpha_i > 0$). Another method is to solve the following:

$$
b = -\frac{\max_{y_i=-1}\left(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle\right) + \max_{y_i=+1}\left(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle\right)}{2} \tag{3.12}
$$

Classifying a novel vector becomes a simple matter of calculating:

$$
f(\mathbf{x}) = \operatorname{sgn}\left(\mathbf{w} \cdot \mathbf{x} + b\right) \tag{3.13}
$$

3.3.2 Non-separable Case

Up to now, it was assumed that the training vectors are linearly separable. This assumption rarely holds in real world problems. Thankfully, there is a solution that was provided by Cortes & Vapnik in 1995 [31]. By introducing slack variables, the constraints given in Eq. 3.2 can be relaxed. The new constraints are as follows:

$$
\begin{aligned}
&\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b \geq +1 - \xi_i \qquad \{i \mid y_i = +1\} \\
&\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b \leq -1 + \xi_i \qquad \{i \mid y_i = -1\} \\
&\xi_i \geq 0 \qquad\qquad \forall i
\end{aligned}
\tag{3.14}
$$

The goal is to keep slack variables as small as possible. A value greater than one implies a misclassification. The sum of slack variables is, therefore, an upper bound on the number of classification errors. The original objective function $\|w\|^2/2$ is augmented with the penalty factor for the slack variables as follows to form

$$\|w\|^2/2 + C\left(\sum_{i=1}^{l}\xi_i\right). \tag{3.15}$$

where C is user modifiable. Higher values of C put a higher penalty on misclassifications. The derivation of the dual Lagrange problem is similar to that above. Details can be found in [32]. The new dual optimization problem becomes:

$$L_D \equiv \sum_{i=1}^{l}\alpha_i - \frac{1}{2}\sum_{(i,j)=(1,1)}^{(l,l)} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle \tag{3.16}$$

With the constraints:

$$\begin{aligned} &0 \le \alpha_i \le C, \qquad \forall i \\ &\sum_{i=1}^{l}\alpha_i y_i = 0 \end{aligned} \tag{3.17}$$

The only difference is an upper bound on the α_i's making it into a *box-constrained* problem. Note that the slack variables disappear. For reference, the new primal Lagrangian is:

$$L_P \equiv \frac{1}{2}\|w\|^2 + \sum_{i=1}^{l}\xi_i - \sum_{i=1}^{l}\alpha_i\left[y_i\left(\langle \mathbf{x}_i \cdot \mathbf{w}\rangle + b\right) - 1 + \xi_i\right] - \sum_{i=1}^{l}\mu_i\xi_i \tag{3.18}$$

in which the μ_i's are new Lagrange multipliers introduced to enforce positivity of the ξ_i's.

The complete KKT conditions for the primal problem are shown below. Note that d is the number of dimensions in the input space, and l is the number of training samples.

$$
\begin{aligned}
&\frac{\partial L_P}{\partial w_v} = w_v - \sum_{i=1}^{l} \alpha_i y_i x_{iv} = 0, && v = 1,\ldots,d \\
&\frac{\partial L_P}{\partial b} = \sum_{i=1}^{l} \alpha_i y_i = 0 \\
&\frac{\partial L_P}{\partial \xi_i} = C - \alpha_i - \mu_i = 0, && \forall i \\
&y_i\left(\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b\right) - 1 \geq 0, && i = 1,\ldots,l \\
&\xi_i \geq 0, && \forall i \\
&\alpha_i \geq 0, && \forall i \\
&\mu_i \geq 0, && \forall i \\
&\alpha_i \left[y_i\left(\langle \mathbf{x}_i \cdot \mathbf{w} \rangle + b\right) - 1 + \xi_i \right] = 0, && \forall i \\
&\mu_i \xi_i = 0, && \forall i
\end{aligned}
\tag{3.19}
$$

The higher complexity of the primal Lagrangian with the addition of slack variables further validates the preference for solving the simpler Wolfe dual problem. From this point on, only the dual formulation will be considered.

3.3.3 Non-linear Case

The Wolfe dual can be rewritten to conform to the representation of a linearly constrained quadratic programming problem:

$$
\begin{aligned}
\psi(\alpha) &= \frac{1}{2}\overline{\alpha}^T G \overline{\alpha} - \sum_{i=1}^{n} \alpha_i \\
&= \frac{1}{2}\overline{\alpha}^T G \overline{\alpha} - \overline{c}^T \overline{\alpha}
\end{aligned}
\tag{3.20}
$$

in which *c* is a vector of all *-1*'s. The problem solution is subject to conditions:

$$
\begin{aligned}
&0 \leq \alpha_i \leq C, && i = 1,\ldots,n \\
&\sum_{i=1}^{n} \alpha_i y_i = 0
\end{aligned}
\tag{3.21}
$$

in which G takes on the values $a_{i,j} = y_i y_j x_i, x_j$.

In order to extend the application of SVMs to nonlinear problems, [29] applied an old kernel technique, known as *kernel-induced spaces,* in which they replaced the values of the matrix *G* with $a_{i,j} = y_i y_j K\left(x_i, x_j\right)$. In effect, they replaced dot product operations with a *kernel function* $K(a,b)$ - hence the name *kernel matrix* for *G*. The *kernel function* may be nonlinear and often performs a transformation onto a higher-dimensional space in which the dot product is performed. Common kernel functions are:

Polynomial (homogeneous)

$$K(\mathbf{a},\mathbf{b}) = \langle \mathbf{a} \cdot \mathbf{b} \rangle^d$$

Polynomial (inhomogeneous)

$$K(\mathbf{a},\mathbf{b}) = (\langle \mathbf{a} \cdot \mathbf{b} \rangle + 1)^d$$

Radial Basis Function

$$K(\mathbf{a},\mathbf{b}) = \exp\left(-\gamma \|\mathbf{a}-\mathbf{b}\|^2\right), \text{ for } \gamma > 0 .$$

Gaussian Radial Basis Function

$$K(\mathbf{a},\mathbf{b}) = \exp\left(-\frac{\|\mathbf{a}-\mathbf{b}\|^2}{2\sigma^2}\right)$$

Sigmoid

$$K(\mathbf{a},\mathbf{b}) = \tanh\left(\langle \kappa \mathbf{a} \cdot \mathbf{b} \rangle + c\right), \text{ for at least one } \kappa > 0 \text{ and } c < 0 .$$

The kernel operation is simplified before implementation whenever possible. For example, in the case of the inhomogeneous polynomial kernel:

$$\begin{aligned} K(x,z) &= \left(\langle \mathbf{a} \cdot \mathbf{b} \rangle + c\right)^2 \\ &= \left(\sum_{i=1}^{l} a_i b_i + c\right)\left(\sum_{j=1}^{n} a_j b_j + c\right) \\ &= \sum_{i=1}^{l}\sum_{j=1}^{l} a_i a_j b_i b_j + 2c\sum_{i=1}^{n} a_i b_i + c^2 \\ &= \sum_{(i,j)=(1,1)}^{(l,l)} (a_i a_j)(b_i b_j) + \sum_{i=1}^{n} \left(\sqrt{2ca_i}\right)\left(\sqrt{2cb_i}\right) + c^2 \end{aligned} \tag{3.22}$$

Since only the kernel matrix is affected by the input vectors, the kernel function decouples the data from the problem. The quadratic program problem is solved in exactly the same way, but with a different, usually larger kernel. More often than not, mapping the input vectors using the nonlinear kernel makes them linearly separable in the induced feature space. The classification step is modified in the same way – with dot products replaced by kernel operations.

3.4 Implementations

As shown in the previous section, the underlying operation for training of SVMs is finding the solution to a linearly constrained quadratic programming problem. The

ramifications of this discovery were significant in that it was possible to apply existing well-developed solvers.

Unfortunately, existing solvers act on the assumption that the entire kernel matrix is in fast memory (usually RAM) at all times. The size of this matrix in the context of SVMs grows quadratically with the number of input training vectors. With number of training vectors in the hundreds of thousands being not uncommon, the memory requirement for the training from such datasets becomes too large. Even with the rapid increase in memory capacity in modern hardware, the poor scalability to increased training set sizes called for new, memory efficient methods. Two such methods which gained wide acceptance are the *Working Set* technique (e.g.: Chunking and Decomposition) and the *Sequential Minimal Optimization* technique.

3.4.1 Working Sets

The main concern, as expressed above, is the lack of memory for the kernel matrix due to the large size of training sets. The working set technique, introduced by Osuna [], attempts to remedy this issue by working on a relatively small subset of the training set at a time. The generalized pseudo-code is shown in Listing 3.

```
Initialization
- given training set S
- select set Ŝ from S
- α ← 0

repeat
        generate problem data based on Ŝ (i.e.: kernel matrix)
        run QP optimizer on the subproblem
        select new working set Ŝ based on KKT violations
until stopping criterion satisfied
return α
```

Listing 3: Generic Working Set Pseudocode

Initially, the subset of size *N* is chosen arbitrarily, usually randomly and evenly distributed. The quadratic programming solver is put to work on this subset and produces some set of α_i's. Those training vectors for which α_i is zero (non-support vectors) are discarded, and in their place *M* new vectors are included based on the magnitude of their KKT violations in the most-current solution. As the algorithm continues, non-support vectors are eliminated until only support vectors are left.

Chunking does not completely eliminate the memory problem as there is no guarantee that the final number of support vectors does not produce a kernel matrix of a size that cannot fit into memory. A more complex method which also falls into the working set method is the *Decomposition* technique. The main difference in the decomposition algorithm is that the size of the working set stays constant and does not exceed the limit of the hardware's available memory. Only those α_i's which correspond to training vectors within the current working set are modified. All others are kept fixed. This

method optimizes the problem by working on a small subset at a time rather than trying to find all the constraints and optimizing on all of them at once.

These algorithms have not been proven to be optimal, but they have shown very good results. The decomposition technique will be further explored in the *Gradient Projection-based Decomposition Technique* section in this chapter.

3.4.2 Sequential Minimal Optimization

The Sequential Minimal Optimization (SMO) algorithm is an extreme case of the Chunking technique, as it analyzes exactly two training vectors at a time – the smallest amount possible. This method is very attractive for many reasons. Having only two free variables allows the use of analytical methods instead of complex QP solvers. This property makes this algorithm much easier to implement and requires less computational and storage resources. While generally needing many more iterations for convergence, theoretically convergence time may be cut by up to a thousand times [33].

Another advantage of the SMO algorithm is that it does not utilize a kernel matrix. Not only does this relax memory requirements, but it is much less susceptible to precision errors due to the nature of floating point operations on modern hardware.

SMO consists of two main parts: the analytic method for updating the two Lagrange multipliers, and a method for choosing the next two variables for update.

The analytical method is based on the dual Lagrange formulation. The original constrains are given in Eq. 3.21.

It follows from the first condition that given two free Lagrange multipliers, both need to be modified in such a way that they lie on the line:

(3.23) $$\alpha_{1new} y_1 + \alpha_{2new} y_2 = c = \alpha_{1old} y_1 + \alpha_{2old} y_2$$

The second condition bounds the values into a box. The freedom for choice of the two Lagrange multipliers is represented graphically in Fig. 3-2. There are two cases: one in which $y_1 = y_2$ (Fig. 3-2a) and one in which $y_1 \neq y_2$ (Fig. 3-2b).

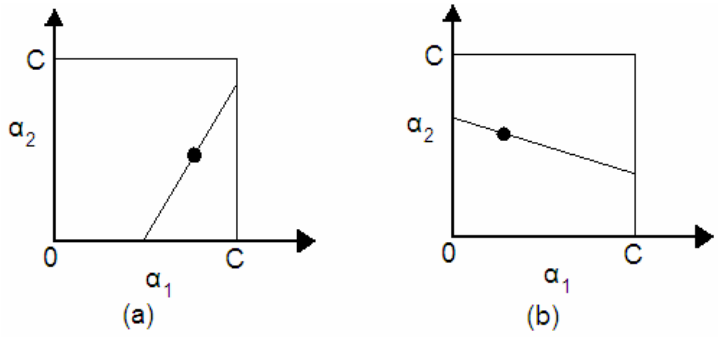


Figure 3-2: Box constrained search

The value of α_2 is calculated first. Analytically, the bounds are:

$$
\begin{aligned}
&L = \max(0, \alpha_2 - \alpha_1) \\
&H = \min(C, C + \alpha_2 - \alpha_1) \\
&\text{if } y_1 = y_2 \text{ and} \\
&L = \max(0, \alpha_2 + \alpha_1 - C) \\
&H = \min(C, \alpha_2 + \alpha_1) \\
&\text{if } y_1 \neq y_2
\end{aligned}
\tag{3.24}
$$

In order to provide the algorithm for calculating the new Lagrange multipliers, two mathematical definitions are needed:

$$
\begin{aligned}
\eta &= K(\mathbf{x}_1, \mathbf{x}_1) + K(\mathbf{x}_2, \mathbf{x}_2) - 2K(\mathbf{x}_1, \mathbf{x}_2) \\
E_i &= u_i - y_i
\end{aligned}
\tag{3.25}
$$

η is the second derivative of the objective function along the diagonal and u_i is the actual output from the classifier.

Calculating α_{2new} and α_{1new} is performed as shown in Eq. 3.27 and 3.28 respectively.

$$
\alpha_{2new,unclipped} = \alpha_2 + \frac{y_2(E_1 - E_2)}{\eta}
\tag{3.26}
$$

$$
\alpha_{2new} = \begin{cases} H & \text{if} \quad \alpha_{2new,unclipped} \geq H \\ \alpha_{2new,unclipped} & \text{if} \quad L < \alpha_{2new,unclipped} < H \\ L & \text{if} \quad \alpha_{2new,unclipped} \leq L \end{cases}
$$

$$
\alpha_{1new} = \alpha_1 + y_1 y_2 (\alpha_2 - \alpha_{2new})
\tag{3.27}
$$

It is important to note that for this technique to work, the chosen kernel function needs to satisfy Mercer's condition [32] and that no two training vectors are equal in the set. The first necessity guarantees a positive η and the second guarantees a non-zero η. There is a method to work around the positivity requirement that requires more computations [33], but it is beyond the scope of this thesis.

The second part of the SMO algorithm is used for selecting the next pair of Lagrange multipliers which are to be modified. According to Osuna's theorem [34], as long as one

of the chosen multipliers violated the KKT condition, the objective function will decrease and convergence will eventually occur. Convergence is sped up by applying heuristics.

The premise behind the procedure is to focus on KKT violating bound training samples – that is those samples for which $0 < \alpha_i < C$ and the KKT test does not pass. Samples which are bounded $\alpha_i = 0$ or $\alpha_i = C$ are likely to stay bounded, and thus are only checked once all bound samples meet the KKT conditions.

Two samples need to be selected. The outer loop of the heuristic algorithm is responsible for choosing the first sample. Initially, all samples are iterated and those which violate the KKT conditions are processed. The next iterations of the loop only work on the non-bound samples that violate the KKT conditions. The loop continues until all of the non-bound samples obey the KKT conditions within a threshold ε. Once this occurs, the entire set is processed again. The loop continues until all α_i's obey the KKT conditions within the threshold.

For every selection of the first parameter a second one is chosen so that good progress is made in the optimization function (large step size). This step size can be approximated by taking the absolute values between E_1 and E_2. The E_i values for all the non-bound samples are cached to reduce kernel evaluations.

For more details of the algorithm, including choices for the threshold ε, unusual circumstances and their workarounds, as well as a pseudo code of the entire process, the reader is directed to [33].

3.4.3 Gradient Projection-based Decomposition Technique

The definition of the working set technique functions only as a high level framework for the pieces that make up the algorithm. The choice of QP solver, working set replacement heuristics, termination conditions, etc. is up to the implementer. Research has advanced greatly in the field of these individual sub-algorithms, making this a hot area in the field of machine learning.

The *Gradient Projection-based Decomposition Technique* (GPDT) [35] is an implementation of the decomposition technique which wraps several recent promising works into a self-contained package. The research papers and the C source code implementation of the algorithm, called the Parallel Gradient Projection-based Decomposition Technique (PGPDT) are available online [70]. The source code available was the starting point of the SVM portion of this work. In this section, only those features that are relevant to this work are described.

The GPDT algorithm was designed to be effective on medium-to-large sized working sets ($O(10^2)$ or $O(10^3)$) and is touted as being the first implementation that is well suitable for application on multi-processor systems. Parallelism exists mainly due to the design of the QP Solver and a global gradient updating step used for the replacement algorithm. The QP solver, which solves each of the working set subproblems, is based on a gradient projection method which has a large matrix-vector multiplication operation for the root of

its computation time. As part of the subproblem training set replacement strategy, the gradient of the global objective function must be calculated within each iteration. This step, which involves another matrix-vector multiplication, can also be parallelized. Special steps need to be taken, however, as the matrix is assumed to be out of memory and needs to be generated or recalled from a cache on the fly.

Given n training samples, the top-level algorithm is shown in Listing 4. Note that this is just a more detailed version of Listing 1.

```
Initialization
- given training set S
- given n > n_sp > n_c > 0, n_c even
- select set Ŝ from S, s.t. #{Ŝ} = n_sp
- α ← 0

repeat
        generate problem data based on Ŝ (i.e.: kernel matrix)
        run gradient projection-based QP optimizer on subproblem
        update global gradient vector based on results
        create new working set Ŝ of size n_sp
        - select up to n_c vectors based on gradient values
        - retain some elements of current working set so that #{Ŝ} = n_sp
until stopping criterion satisfied
return α
```

Listing 4: GPDT Algorithm, top level.

Let G, $\boldsymbol{\alpha}$, and $\boldsymbol{y}$ be the global matrix kernel, Lagrange multipliers, and classifications $\{-1,+1\}$ for the entire training set (with G being out of memory), β be the set of training vector indices within the working set of the current iteration, and δ be the complementing set not in the current working set. The decomposition is performed by separating the data into:

$$G = \begin{bmatrix} G_{\beta\beta} & G_{\beta\delta} \\ G_{\delta\beta} & G_{\delta\delta} \end{bmatrix}, \qquad \alpha = \begin{bmatrix} \alpha_\beta \\ \alpha_\delta \end{bmatrix}, \qquad y = \begin{bmatrix} y_\beta \\ y_\delta \end{bmatrix} \tag{3.28}$$

3.4.3.1 QP Solver

The solver used in the GPDT implementation is itself made up of existing well-developed ideas and several modifications added for more efficient convergence. The method is described in detail in [36]. It is based on gradient projection onto a box-constrained space – a method that is relatively trivial to implement. The problem is restated below.

$$\min \quad f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{c}^T\mathbf{x} \tag{3.29}$$

s.t.:

(3.30)
$$\begin{aligned} &\mathbf{a}^T\mathbf{x} = b \\ &L \le \mathbf{x}_i \le U, \qquad\qquad i = 1,...,n \end{aligned}$$

Note that when training SVMs, n is the number of training samples, A is $G_{\beta\beta}$, $\mathbf{c}$ is a vector of ones, L is 0 and U is specified as a training parameter. If the matrix A is positive definite and diagonal $\left(A = diag\left(d_0, d_1, ..., d_n\right)\right)$, the projection step alone is enough to find the solution. Otherwise, a necessary line search and step length calculation step must also be performed for convergence to occur. The projection step will be described first followed by the line search and step size algorithms.

The first step is to represent the problem a different way by transforming the first constraint into a penalty term in the objective function:

(3.31)
$$\phi(\mathbf{x};\lambda) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{c}^T\mathbf{x} - \lambda\left(\mathbf{a}^T\mathbf{x} - b\right)$$

in which λ is a scalar parameter that needs to be found. The portion $\mathbf{a}^T\mathbf{x} - b$ is derived from the constraint $\mathbf{a}^T\mathbf{x} = b$ of the original optimization problem. Next, an educated guess for an initial value for λ is taken resulting in the problem:

(3.32)
$$\begin{aligned} \min \quad & \phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{c}^T\mathbf{x} - \lambda_{const}\left(\mathbf{a}^T\mathbf{x} - b\right) \\ \text{s.t.:} \quad & L \le \mathbf{x}_i \le U, \qquad\qquad i = 1,...,n \end{aligned}.$$

The method for finding the minimizer $\mathbf{x}(\lambda)$ for some λ is very easy as it separates into n separate problems, each with one variable. Each variable x_i is found by solving:

(3.33)
$$d_i x_i = c_i + \lambda a_i .$$

Rearranged, this gives:

(3.34)
$$x_i = \left(c_i + \lambda a\right)_i / d_i .$$

Due to the box constraint, each variable is clamped as shown in Eq. 3.35.

(3.35)
$$x_{i,clamped} = \begin{cases} U & \text{if} \quad x_i \ge U \\ x_i & \text{if} \quad L < x_i < U \\ L & \text{if} \quad x_i \le L \end{cases}$$

Due to clamping, the constraint $\mathbf{a}^T\mathbf{x}(\lambda) - b = 0$ will most likely not be met, and thus λ will need to be readjusted iteratively using an outer secant-like method until the equality in 3.36 holds.

(3.36)
$$r(\lambda) := \mathbf{a}^T\mathbf{x}(\lambda) - b = 0$$

To summarize, the controlled variable is λ, and the output is r which is to be equal to 0 to satisfy the second of the two constraints in the original problem. The value of r is determined by the intermediate solution to the optimization problem 3.36. The task, therefore, is to find the point of intersection on the graph of λ vs. r. This graph represents a monotonically increasing piecewise linear continuous function of λ. However, it is not smooth, and therefore common gradient methods cannot be easily applied. Instead, a modified secant-like method is used. This algorithm, from here on referred to as Algorithm 1, is divided into two parts: the bracketing phase and the secant phase.

The bracketing phase is designed to find a minimum and maximum of the λ between which the solution lies. The pseudo code is shown in Listing 5.

Calculate $\mathbf{x}$ *by (3.34) and (3.35)*
$r = a^T x - b$
if($r < 0$ *)*
    $\lambda_l = \lambda; r_l = r; \lambda = \lambda + \Delta\lambda$
    Calculate $\mathbf{x}$ *by (3.34) and (3.35)*
    $r = a^T x - b$
    while($r < 0$ *)*
        $\lambda_l = \lambda; r_l = r; s = \max(r_l / r - 1, 0.1)$
        $\Delta\lambda = \Delta\lambda + \Delta\lambda / s; \lambda = \lambda + \Delta\lambda$
        Calculate $\mathbf{x}$ *by (3.34) and (3.35)*
        $r = a^T x - b$
    end
    $\lambda_u = \lambda; r_u = r$
else
    $\lambda_u = \lambda; r_u = r; \lambda = \lambda - \Delta\lambda$
    Calculate $\mathbf{x}$ *by (3.34) and (3.35)*
    $r = a^T x - b$
    while($r > 0$ *)*
        $\lambda_u = \lambda; r_u = r; s = \max(r_u / r - 1, 0.1)$
        $\Delta\lambda = \Delta\lambda + \Delta\lambda / s; \lambda = \lambda - \Delta\lambda$
        Calculate $\mathbf{x}$ *by (3.34) and (3.35)*
        $r = a^T x - b$
    end
    $\lambda_l = \lambda; r_l = r$
end

Listing 5: Bracketing phase of Algorithm 1

The modified secant search step searches within this space until the solution is found. Its pseudo code is shown in Listing 6.

$s = 1 - (r_l / r_u); \Delta\lambda = \Delta\lambda / s;\ \lambda = \lambda - \Delta\lambda$

Calculate $\mathbf{x}$ *by (3.34) and (3.35)*

$r = a^T x - b$

while not converged

  if(r > 0)

    if(s <= 2)

      $\lambda_u = \lambda; r_u = r; s = 1 - r_l / r_u$

      $\Delta\lambda = (\lambda_u - \lambda_l)/s; \lambda = \lambda_u - \Delta\lambda$

    else

      $s = \max(r_u / r - 1, 0.1); \Delta\lambda = (\lambda_u - \lambda)/s$

      $\lambda_{new} = \max(\lambda - \Delta\lambda, 0.75\lambda_l + 0.25\lambda)$

      $\lambda_u = \lambda; r_u = r; \lambda = \lambda_{new}$

      $s = (\lambda_u - \lambda_l)/(\lambda_u - \lambda)$

    end

  else

    if(s >= 2)

      $\lambda_l = \lambda; r_l = r; s = 1 - r_l / r_u$

      $\Delta\lambda = (\lambda_u - \lambda_l)/s; \lambda = \lambda_u - \Delta\lambda$

    else

      $s = \max(r_l / r - 1, 0.1); \Delta\lambda = (\lambda - \lambda_l)/s$

      $\lambda_{new} = \min(\lambda + \Delta\lambda, 0.75\lambda_{ul} + 0.25\lambda)$

      $\lambda_l = \lambda; r_l = r; \lambda = \lambda_{new}$

      $s = (\lambda_u - \lambda_l)/(\lambda_u - \lambda)$

    end

  end

  Calculate $\mathbf{x}$ *by (3.34) and (3.35)*

  $r = a^T x - b$

end

Listing 6: Secand phase of Algorithm 1

As mentioned, the projection method on its own is only valid for problems in which matrix *A* is positive-definite and *diagonal* – a case that is highly unlikely in the case of SVMs. It is possible to extend the algorithm to the general non-diagonal case by encapsulating the algorithm into an outer loop along with the *linesearch* and *step length* algorithms that were mentioned previously. This new framework is similar to that in [37] and is shown in Listing 7.

```
initialization
repeat
        Calculate Projection using Alg 1
        Possibly carry out a line search
        Calculate BB-like step length
        Update the line search control parameters
until stopping criterion satisfied
```

Listing 7: Outer Loop for QP solver allowing for non-diagonal matrices A.

The first step in the loop uses Algorithm 1 to take the steepest descent step from the current location $\mathbf{x}_k$ with a fixed step length and project the result onto the feasible space as defined by the constrains in the original problem. The result of this operation gives a feasible step direction. This is shown mathematically as

$$\mathbf{d}_k = P_\Omega\left(\mathbf{x}_k - \alpha_k \mathbf{g}_k\right) - \mathbf{x}_k \tag{3.37}$$

in which $\mathbf{g}_k = A\mathbf{x}_k - \mathbf{c}$ and $P_\Omega(\mathbf{z})$ is the projection of a vector $\mathbf{z}$ onto the feasible space Ω.

The second step of the algorithm is to carry out a line search along the step direction. This step is not required when minimizing unconstrained quadratic problems, but it has been shown that the algorithm may fail in the constrained case [38]. The reasoning for including the line search step is that the step length calculation, which is the next part of the algorithm, relies on the steady increase of the objective function $f(\mathbf{x})$ in order to work properly.

The line search algorithm only performs the search when necessary, thus cutting down on the computations needed. It relies on a control parameter f_{ref}, which along with certain other parameters, is dynamically updated in the final step in the main loop. The line search step executes only if $f(\mathbf{x}+\mathbf{d}) \geq f_{ref}$. The process is carried out by quadratic interpolation along $\mathbf{x}+\lambda\mathbf{d}$ by using the objective function values of $f(\mathbf{x})$ and $f(\mathbf{x}+\mathbf{d})$ and the slope $\mathbf{g}^T\mathbf{d}$.

The third step of the algorithm uses a modified Barzilai-Borwein (BB) step size formula for computing the step size to take. The BB steepest descent method has been used extensively for large-scale optimization problems. Fletcher gives an overview of the recent developments in the application to large scale unconstrained optimization in [39]. The choice for selecting the step size in this algorithm is:

$$\alpha_{k+1} = \frac{\sum_{i=0}^{m-1} \mathbf{s}_{k-1}^T \mathbf{s}_{k-1}}{\sum_{i=0}^{m-1} \mathbf{s}_{k-1}^T \mathbf{y}_{k-1}} \tag{3.38}$$

In which:

$$\begin{aligned} \mathbf{s}_k &= \mathbf{x}_{k+1} - \mathbf{x}_k \\ \mathbf{y}_k &= \mathbf{g}_{k+1} - \mathbf{g}_k \end{aligned} \tag{3.39}$$

In the formula above, m is a preset integer, with 2 being a common value. A new variable $\overline{m}$ is defined to be the maximal integer for which $\mathbf{s}_{k-1}^T \mathbf{y}_{k-1} > 0$ for all $0 \leq i \leq \overline{m}$. In the implementation, m is replaced by $\min(m, \overline{m})$. The resulting step size is then clamped between two extremes $[\alpha_{\min}, \alpha_{\max}]$. If $\overline{m}$ is 0, the value $\alpha_{\max}$ is used.

The last step in the algorithm is the updating of the control parameters for the line search algorithm. These parameters are f_{ref}, described above, a candidate value f_c for possible reduction of f_{ref}, the current best value f_{best}, a counter l of consecutive iterations during which $f(x) \geq f_{best}$, and the number L of such iterations allowed before reducing f_{ref} to f_c. The algorithm is shown in Listing 8.

```
if ( f_k < f_best )
        f_best      = f_k
        f_c         = f_k
        l           = 0
else
        f_c         = max( f_c, f_k )
        l           = l + 1
        if ( l == L )
                f_ref       = f_c
                f_c         = f_k
                l           = 0
        endif
endif
```

Listing 8: Updating line search control parameters

The description and reasoning for this algorithm can be found in [36], and [38].

Convergence of the algorithm is monitored during the projection step. The solution to the problem occurs when $P_\Omega(\mathbf{x}_k - \mathbf{g}_k) = \mathbf{x}_k$; in other words, when $P_\Omega(\mathbf{x}_k - \mathbf{g}_k) - \mathbf{x}_k = 0$. In practice, a threshold value is used instead of 0. To prevent an extra projection step, given a proper size of a_k an alternative calculation of $\|d\|/a_k$ may be used to similar effect.

In the full algorithm, the most computationally expensive part is the calculation of the gradient $\mathbf{g}_k = A\mathbf{x}_k - \mathbf{c}$. This matrix-vector operation is easily parallelizable. Initializing the subproblem matrix is also expensive due to the kernel calculations required. Depending on the low level implementation, this step can also be optimized for parallel systems. The procedure of doing so in this work is described in Chapter 7.

For further details on this algorithm, common alterations, as well as workarounds to the case that the original constrained problem is not solvable, the reader is invited to refer to [36].

3.4.3.2 Updating the Global Gradient

The QP solver described above can only function on problems small enough that they fit into the available memory. After each subproblem solution, it is necessary to update the global gradient whose dimension is the size of the global set. The gradient is necessary so that the elements for the subsequent working set can be efficiently selected. This step is similar to the step within the QP solver – namely the calculation of $\mathbf{g}_k = A\mathbf{x}_k - \mathbf{c}$ before projecting into the feasible space, with the exception that A is most likely out of memory and unavailable and thus its elements must be calculated on the fly.

The large number of kernel operations typically required in this step tends to make it the most computationally expensive in the entire algorithm. One way in which the PGPDT implementation reduces the required kernel operations is by introducing a least recently used caching strategy (which is carried over to the implementation in this work). The program accepts a command line parameter specifying the amount of memory to reserve for the cache which stores the most recently calculated columns of the global kernel matrix. The other trick that PGPDT utilizes is the processing of those input vectors within the subproblem for which α changed by more than a small threshold.

Let N_p be the total number of training examples and G_i denote the i^{th} column of the global kernel matrix. Let

$$\beta_{gu} = \{i \in \beta \mid abs(\alpha_{k+1} - \alpha_k) > sv0\} \tag{3.40}$$

in which *sv0* is a value close to 0, G_{gu} be the kernel matrix including only column indices from the set β_{gu}. The gradient is updated using:

$$\nabla F(\alpha_{k+1}) = \nabla F(\alpha_k) + G_{gu}(\alpha_{k+1} - \alpha_k). \tag{3.41}$$

Note, that for simplicity, the cache strategy, as outlined in the original paper [35] has been omitted. The implementation in this work differs, and will be detailed in chapter 7.

3.4.3.3 Working Set Replacement Strategy

For the final step of the algorithm, a new working set must be chosen for the following iteration. The selection procedure was introduced in [40] and tested in [41]. The procedure is to solve the problem:

$$\begin{aligned}
\min \quad & \nabla F(\alpha_{k+1})^T d \\
s.t.: \quad & y^T d = 0 \\
& d_i \geq 0 \ \text{ for i such that } \alpha_{i,k+1} = 0 \\
& d_i \leq 0 \ \text{ for i such that } \alpha_{i,k+1} = C \\
& -1 \leq d_i \leq 1 \\
& \#\{d_i \mid d_i \neq 0\} \leq n_c
\end{aligned}$$

This portion was not modified in any way within the PGPDT implementation as it was deemed to be insignificant in the overall computation time. The procedure for the solution of this problem was implemented as shown in Listing 9.

Sort the indices of the variables according to $y_i \nabla F(\alpha_{k+1})_i$ *in decreasing order and let* $I \equiv (i_1, i_2, ..., i_n)^T$ *be the sorted list.*

Repeat the selection of a pair $(i_b, i_t) \in I \times I$, with $t < b$, *as follows:*

- moving down from the top of the sorted list, choose $i_t \in I_{top}(\alpha_{k+1})$

- moving up from the bottom of the sorted list, choose $i_t \in I_{top}(\alpha_{k+1})$

until n_c *indices are selected or a pair with the above properties cannot be found*

Let $\hat{\beta}$ *be the set of these selected indices*

Fill $\hat{\beta}$ *up to* n_{sp} *entries by adding the most recent (least amount of consecutive appearances in the working set and currently in the working set) indices* $j \in \beta$ *satisfying* $0 < \alpha_{j,k+1} < C$*; if these indices are not enough, then add the most recent indices* $j \in \beta$ *such that* $\alpha_{j,k+1} = 0$*, and eventually. the most recent indices* $j \in \beta$ *satisfying* $\alpha_{j,k+1} = C$.

Set $n_c = \min\{n_c, \max\{10, J, n_{new}\}\}$*, where J is the largest even integer such that* $J \le n_{sp}/10$ *and* n_{new} *is the largest even integer such that* $n_{new} \le \#\{j, j \in \hat{\beta} \setminus \beta\}$*; set* $\beta = \hat{\beta}, k \leftarrow k+1$.

Listing 9: Working Set Selection Algorithm

3.4.4 The Cascade SVM

The recent advancement and wider availability of parallel architectures and systems has placed new emphasis on research toward parallel-friendly implementations for many existing algorithms in scientific literature. In the case of the SVM, the working set methods, including chunking and decomposition, are not suited for high-level parallelization due to the dependencies between the major steps of the algorithm. The synchronous nature of the framework limits parallelization to only within the sub-algorithms themselves, as exhibited in the PGPDT design described in the previous section. The cascade method [42] is one new framework which promotes asynchronous execution of independent subproblems generated from a global set. While these algorithms are not optimal, the introduction of several new theorems and proofs argues for an acceptable level of correctness for some applications. By taking advantage of modern parallel hardware, the increased speed of convergence makes these algorithms very attractive when perfect accuracy is not required.

The concept of the Cascade SVM is the construction of a tree of SVM solvers, as shown in Fig. 3.3, with results generated by any one solver being consumed by the parent node. The left-most children of the cascade are given some portion of the global training vectors as inputs. The portions may be exclusive, or may repeat. Each SVM functions as a filter by producing an output consisting of only support vectors and their corresponding

alpha values. Groups of outputs of the SVMs in the first layer are combined using the union operation and used as inputs for the next, smaller, layer of SVMs which perform more filtering. The process continues until the final node (root node) is reached. The exact combination and network rules are not defined, but several theorems exist [43] [42] that should be followed for proper operation. The cascade framework has several attractive properties. Each layer in the network is guaranteed to advance the optimization function, the level of communication is minimal between the layers, and convergence has been shown to be relatively quick. Examples of existing published works follow.

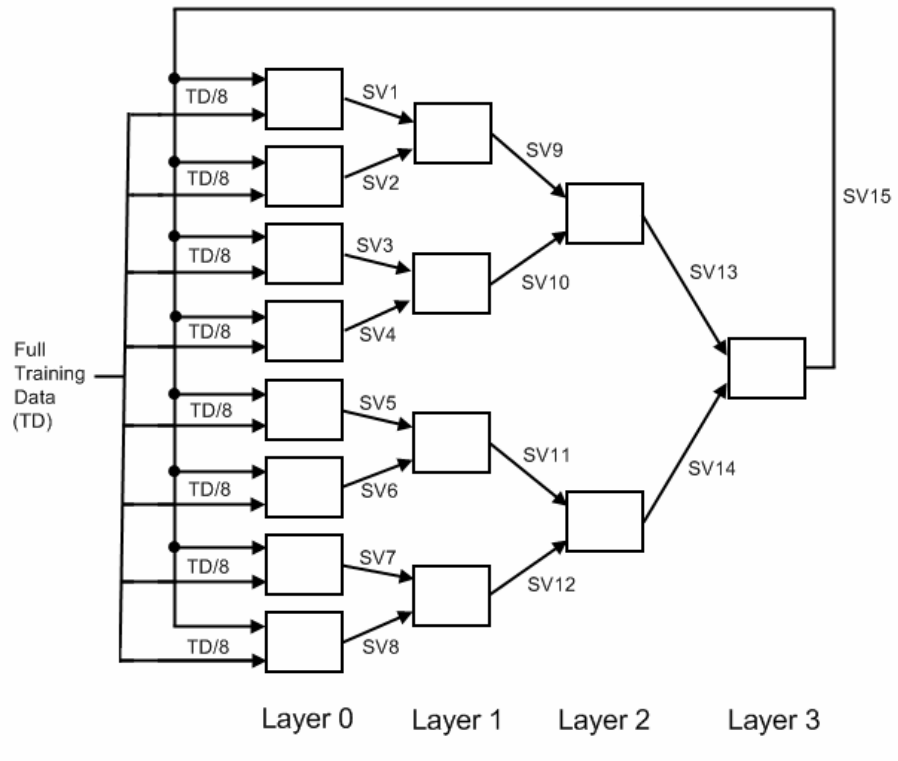


Figure 3-3: Cascade SVM Network

In [42], the authors created a binary network, as shown in Fig. 3-3. At the first layer, all training vectors are split evenly and exclusively for each of the solvers. Results from the solvers are combined in pairs and input into a solver in the next layer. Thus there is exactly one solver for every two solvers in the previous layer. The premise is that vectors eliminated from a subset of the global set are unlikely to be support vectors in the global set. This idea is shown graphically using a 2-dimensional feature space in Fig. 3.3.

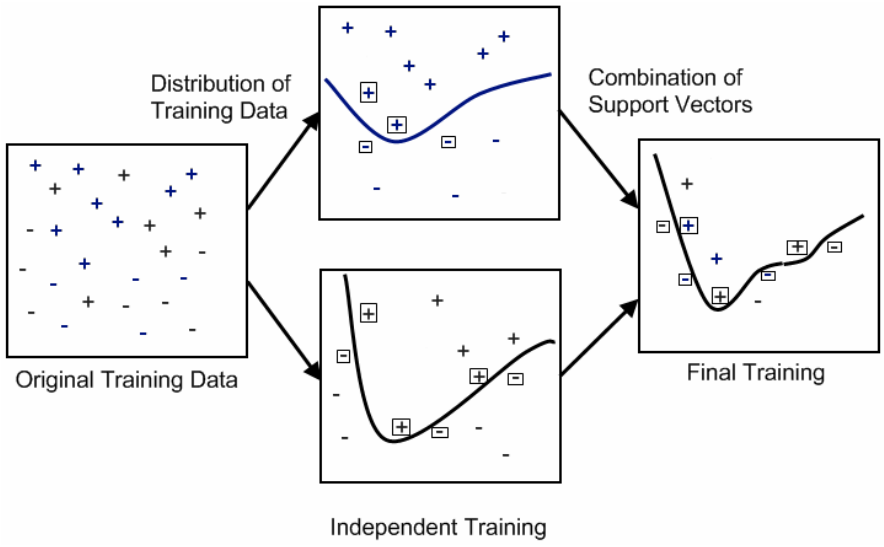


Figure 3-4: Cascade SVM training subset concept

The output from each solver is a subset of the input vectors along with each output vector's nonzero alpha value. There are several ways to combine outputs from multiple solvers for use as input into a new solver. In [42], Graf et al. look at two possibilities. In the first, alpha values from one are combined with those of the second. In the other, alpha values generated by the second SVM are discarded and set to 0. In both situations, the condition $\mathbf{a}^T\mathbf{y} = 0$ holds. The first option should be used when the two solvers operate on completely independent vectors. The other extreme is the case that both solvers are operating on exactly the same vectors, in which the second option should be used. In general, the optimal option is somewhere in between.

The theories presented in the research suggest that the global optimum can be reached if the best set of support vectors produced in one layer is used in at least one of the subsets of the next layer. The binary architecture shown in Fig. 3-3 accomplishes that by providing a feedback link in which the vectors produced by the final solver are combined with each of the original inputs and the process is repeated. The number of passes required, according to their results, is around 2 to 5.

Similar networks, also based on the Cascade SVM method, have been proposed in [44] and [45].

In the M^3-SVM, initial input vectors are first divided into two sets: one with only positively classified training vectors, and one with only negatively classified training vectors. The resulting positive set is randomly divided into $N+$ equally sized subsets and the negative set is randomly divided into $N-$ equally sized subsets. A total of $N+ * N-$ problems are generated by generating every possible pair of a positive and negative set. A problem $T_{i,j}$ is one that includes the i^{th} positive subset and j^{th} negative subset. These subproblems are used as the initial problems and determine the number of leaf nodes in the cascade network. The leaf nodes can be further subdivided if necessary. The authors recommend this step if a good processing load balance is desired.

Two mathematical constructs – the *MIN* and *MAX* mathematical integration units – are defined as:

$$MIN := q = \min(p_1, p_2, ..., p_n)$$
$$MAX := q = \max(p_1, p_2, ..., p_n)$$

These two units are used to integrate the resulting transfer functions to obtain the overall transfer functions. First, all subproblems are grouped according to common positive subsets. Each of these groups is then integrated using a *MIN* unit and one transfer is produced for each group. The resulting transfer functions are integrated using the *MAX* unit, producing the final transfer function. The architecture is displayed in Fig. 3-5.

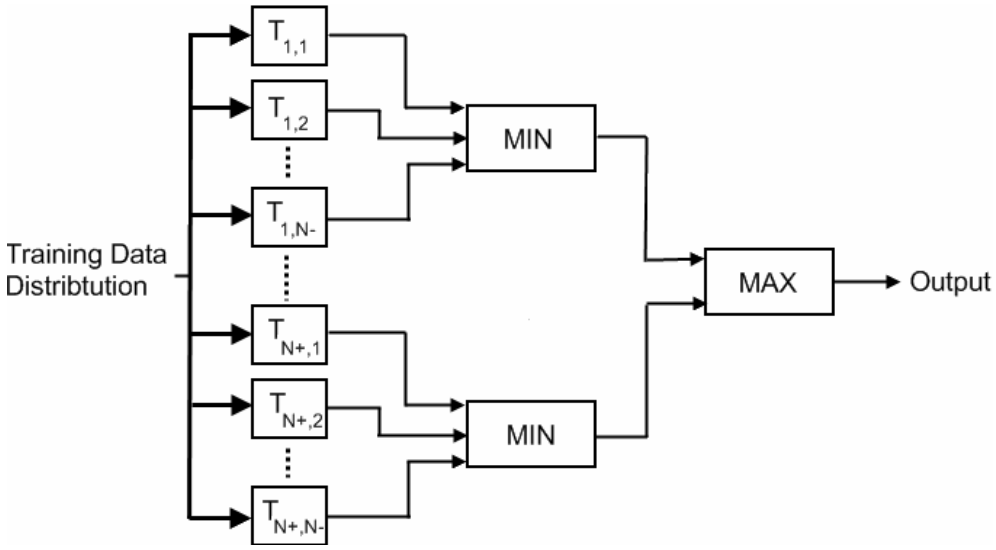


Figure 3-5: M^3-SVM

Lu et al. also designed a standard cascade SVM, but introduced a new method input vector selection in generating subproblems for each of the SVMs in the cascade tree. Their network is shown in Fig. 3-6. First, both the positive classified training vectors and negative classified training vectors are divided into two subsets using the same ratio r ($0 < r < 1$). Next, the four initial subproblems are generated by choosing all four possible combinations of selecting from one of the two positive subsets and the two negative subsets.

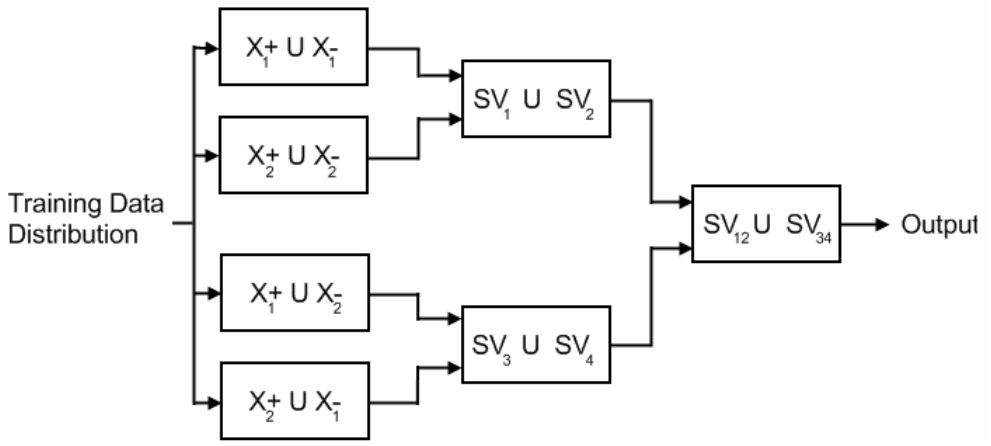


Figure 3-6: Cascade SVM by Lu et. al.

The training vectors for the two intermediate SVMs are chosen by performing a union operation on the resulting support vectors from the corresponding children nodes. Similarly, the resulting vectors from the second layer are combined using the union operator and used to generate the subproblem for the final SVM. The algorithm suggests that alpha values are discarded on the completion of a training process. Only the input vectors are needed to generate new problems. Again, the leaf node problems can be further subdivided if necessary.

The authors also proposed a slightly improved version of their cascade architecture by removing the two intermediate SVMs and combine the output support vectors from the first layer by performing the union operation on all four SVMs. The revised version is shown in Fig. 3-7.

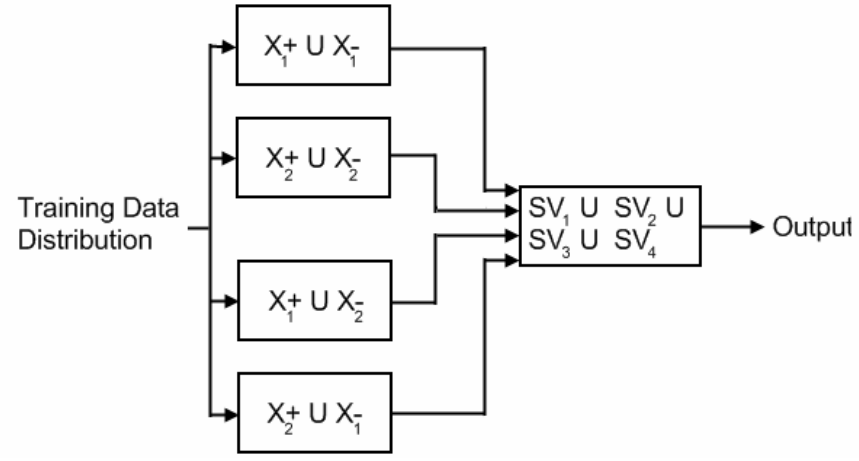


Figure 3-7: Improved Cascade SVM

3.5 Conclusion

Support Vector Machine training can be performed by solving the Quadratic Programming Optimization Problem. Using existing solvers is not possible, however, when very large training data sets are used. The working set technique was developed to

overcome this problem. The Gradient Projection-Based Decomposition Technique, a derivative of the working set technique has not only shown to be highly efficient with large problem sets, but it also has the attractive property of being easily parallelized. The Cascade SVM is a radically new method that was designed for parallelism from the start. Due to the novelty of these techniques, the applicability of them to modern parallel hardware is ripe for exploration.

Chapter 4: The Cell Broadband Engine

4.1 Chapter Introduction

The Cell Broadband Engine Architecture (CBEA) has been designed by Sony, Toshiba, and IBM (STI alliance) in an effort to fill a void between general purpose processors, such as the AMD and Intel family of desktop/laptop processors, and specialized processors such as graphics processing units made by Nvidia and ATI. The flexibility of such a system is very beneficial in many modern media-rich applications.

The new architecture, an extension of the PowerPC Architecture, defines one or more Power Processing Elements (PPEs) and multiple high performance Synergistic Processing Elements (SPEs). The philosophy of the design gives the PPE the role of managing and employing the available SPEs for performing computationally intensive work. The PPE, which is based on the existing IBM Power Architecture, is capable of running existing unmodified 64-bit and 32-bit applications and operating systems, but doing so takes absolutely no advantage of the extra power available. Applications must be written or rewritten to take advantage of the extra cores available.

The Playstation 3 (PS3) System released by Sony in 2006 is one of the first CBEA implementations made available on the market and, due to its use in this work, will be focused on in this chapter.

4.2 Design Challenges

Multimedia performance has always been limited by the problem of unacceptable memory latency and bandwidth (known as the memory wall) as well as problems of diminishing returns arising from increasing the pipeline depth and decreasing the work done per cycle. It has been widely known that memory simply cannot keep up with the rapid increase in CPU performance. Various tricks have been implemented with the goal of hiding this latency, but all come at a hefty price in transistor count, circuit complexity, and power consumption (e.g.: speculative instructions and branch prediction logic). The first design challenge, therefore, was to come up with a way to allow more memory bandwidth at lower latencies.

Another challenge faced was power efficiency. Modern processors can harness only so much performance per Watt with the current CMOS transistor technology before suffering from heat issues.

The third challenge related to performance was overcoming the diminishing returns of increasing the pipeline depth while maintaining instruction latencies. Long pipeline depths imply more logic for dependency tracking and result in significant penalties for incorrectly predicted branches. A design goal was set to minimize the pipeline depth and maximize the efficiency of issue slots for incoming instructions.

The final product was to be highly responsive and reactive to the outside world. This includes, for example, real-time output to stimulate the gamer, and input required for broadband internet applications. In this respect, the processor was to exercise real-time operations for the workloads demanded from it.

Beyond the Playstation 3 system, the plan was to continue developing the architecture so as to implement it in various other future multimedia devices. This required that the design be flexible and extendable. The hardware needed to be easily modifiable and the software within reach of the software community. A Linux-based software development was planned to be developed concurrently for this purpose.

The three companies determined that no current organizational architecture was capable of the computational power fitting their vision of future multimedia devices. The IBM Research Division was first in line to explore new architectural designs for the proposed processor. Over the course of a half a year, IBM considered a wide range of multi-core organizations borrowing concepts from broadband interconnect entertainment systems to supercomputer structures. Finally, a design was agreed on at the end of 2000.

The design was to be based on the currently available 64-bit Power Architecture (to meet the four year deadline) with a memory flow controller and individual processors, or "cells", which were termed "synergistic" processors. Almost immediately, design commenced on a $400,000,000 start-up budget.

4.3 Top Level Design

The implementation of the Cell on the Playstation 3 features a single PPE and eight SPEs for a total of nine processing elements on the chip, tied together using the Elementary Interconnect Bus (EIB). While a total of eight SPEs are included on the chip, only 6 are available for use by the programmer. It is speculated that one of the SPEs has been purposefully disabled to increase the production yield, and that the other is dedicated for running the software-hardware security-focused interaction layer known as the hypervisor. This SPE runs in a so called isolation mode [46].

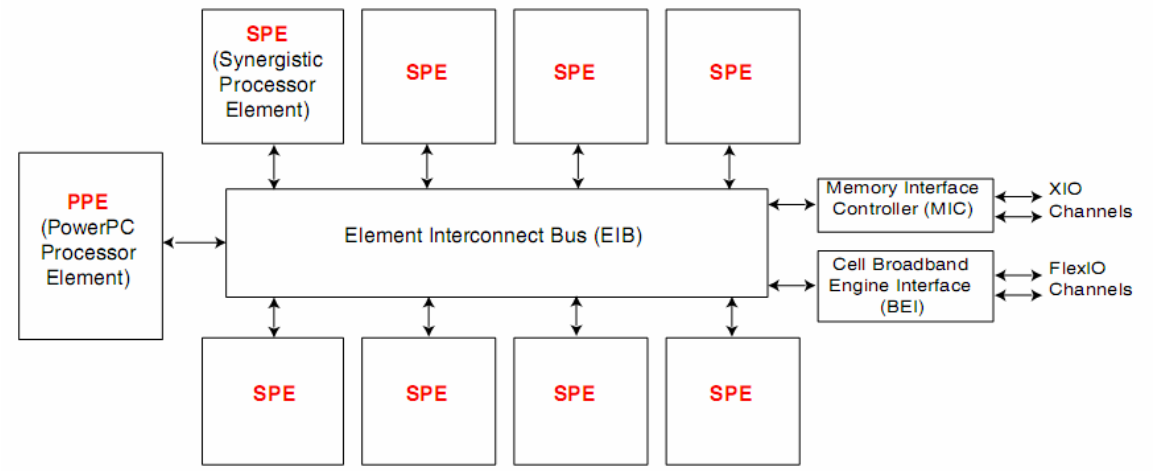


Figure 4-1: Architecture of first iteration of the CBEA
(source: CBE Tutorial v2.1 [60])

The PowerPC based PPE is a fully featured processor capable of running any operating system that supports it. On its own, however, it is easily overpowered even by competing processors of the same generation. It is the remaining 8 SPEs that provide the high degree of processing power, that if optimally programmed, are claimed to outperform specialized systems that are considerably more costly.

A significant design aspect of the CBEA is the concept of independent Local Stores (LS) on each individual SPE. Each SPE is tied to a local LS unit (256 KB of SDRAM on the PS3) which provides it with very fast memory. The contents of the LS are manually controlled by employing the SPE-resident Memory Flow Controller (MFC). Data is transferred between the LS and system memory as well as between multiple LSs via MFC-bound GET and PUT commands. This model provides each SPU with its own memory address space that it can use as an explicitly controlled cache. DMA transfers are globally coherent, meaning that the on-chip cache (on the PPU) may be used transparently.

The entire design strives on parallelism. The PPE includes hardware for two simultaneously executed resource-sharing threads and inherits the Altivec vector instruction set, incorporating both thread-level and instruction-level parallelism respectively. The SPEs feature a dual-issue instruction queue, SIMD (single instruction multiple data) functional units, and a dedicated asynchronously controlled MFC (memory flow controller). Contrary to many existing processor designs, the Cell has been designed around parallelism instead of having it incorporated as an afterthought. Fully exploiting the capabilities of the Cell processor involves not only taking advantage of all of these per-processor capabilities, but also efficiently distributing the workload among the available processing units. The next chapter discusses programming techniques that help in this regard.

In an effort to keep the heat down and clock rates high, and keeping with the original design philosophy, the PPE and SPEs were designed with simplicity in mind. The SPEs

do not include any exotic commodities such as out of order execution or dynamic branch prediction. Logic circuitry is kept to a minimum, shifting the responsibility onto the hands of the compiler and software developers for generating computationally efficient code.

An IBM designed on-chip memory controller is connected to an XIO interface designed by Rambus to keep memory latencies at a minimum. Rambus' flexible and configurable I/O interface, termed FlexIO is used to support the high I/O bandwidth requirement imposed by multimedia applications.

A custom designed on-chip coherent interconnect bus, known as Element Interconnect Bus (EIB) is used to supply the necessary bandwidth required for the nine processors, memory controller, and bus interface. The design of the bus allows for parallel memory transactions as long as the paths do not intercept.

All modules on the Cell are interfaced with the processing elements using memory-mapped control and I/O registers. These registers are grouped into one of three classes: Privilege 1, Privilege 2, or Problem State. Privilege 1 registers are accessed only by the hypervisor or external firmware. Privilege 2 registers are those that should only be accessed by elevated privilege OS modules and are defined for those cases in which a hypervisor is not present. Problem State register access is not enforced by hardware, but may be enforced by the OS or optionally by the hypervisor. The memory mapped registers are used to interface with each of the SPEs, the Pervasive module (used for monitoring performance and temperature, and managing power, and Reliability, Availability, Serviceability debugging), the Memory Interface Controller (MIC) and Token Manager (TKM), the I/O Controller (IOC) Address Translation module, the Bus Interface Controller (BIC), and the Elementary Interconnect Bus (EIB). Details on all these registers are available in [47].

In addition to supporting the 64-bit Power Architecture ISA, CBEA-compliant processors inherit the memory translation, protection and SMP coherence model of 64 bit Power processors. Other inclusions are virtualization for allowing the simultaneous running of multiple operating systems, and large page sizes which are beneficial in many multimedia and scientific applications.

The overall design admits several favorable features to the equation. The simplicity of each processor requires a smaller transistor count and thus allows for higher clock frequencies, lower power dissipation, and better computation/Watt ratio. The Power Architecture in the PPE allows for easy transition for programmers already proficient at software development with this ISA. Being a Single-Instruction-Multiple-Data (SIMD) architecture, and supporting vector media extensions along with providing each processor much on-chip memory and many registers only reinforces the chip’s performance capabilities in parallel applications.

4.4 Low Level Design Decisions

The first release of the PS3 featured a Cell processor utilizing a 90 nm SOI processing technology on a 235.48 mm^2 die with each processor tied to a 3.2 GHz clock. At the time

of this writing, a newer 65 nm version (174.61 mm^2 die) has been made available and is used in currently selling PS3s. A 45 nm version of the chip is in the works.

4.5 Power Processing Element

The PPE is a simplified derivative of IBM's previous 64-bit RISC Power Architectures. The CBEA specifications dictate that the PPE module(s) are 64-bit based (with 32-bit compliance) and include a vector/SIMD multimedia extension unit.

The PPE is compliant to the specifications outlined in the PowerPC Architecture Books I [48], II [49], and III [50], with a few pointers. CBEA compliance requires the inclusion of several instructions which are only optional in the PowerPC Architecture. Two instructions from the graphics group in the PowerPC specifications – floating reciprocal estimate single A-form (fres) and Floating reciprocal square-root estimate A-form (*frsqte*) – are required. Also, the Data cache block touch X-form (*dcbt*), is required, which is used by a program to provide a hint as to what data to place into the cache before it is accessed. The PPE also includes the optional Altivec Vector Instruction Set in the Power Architecture which the CBEA makes mandatory. The pipelined hardware logic allows the PPU to perform two double precision operations per clock cycle (6.4 GFLOPS at 3.2 Ghz) or eight single precision operations per clock cycle (25.6 GFLOPS at 3.2 Ghz).

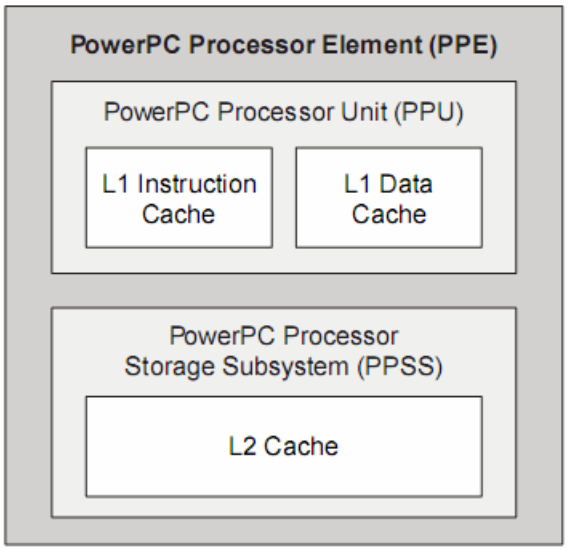


Figure 4-2: PPE Block Diagram (source: Cell Broadband Engine Programming Handbook v1.1 [51])

The PPE consists of the PPU and local L1 and L2 caches. 32 Kb of L1 cache is used for instructions, and 32 Kb is used for data. L2 cache holds both instructions and data and has a size of 512 Kb.

The PPE also supports two-way simultaneous multithreading (SMT), which is very similar to Intel's Hyper-Threading technology, exposing itself as two logical processors to the operating system.

In accordance with the simplicity in the design the processor includes a two-issue in-order core. This approach drastically cuts down on the transistor count, allowing for greater power efficiency and higher clock rates.

The PPE also manages many of the on-chip and off-chip resources for the proper integration with the rest of the system and hence runs the operating system. Hardware resources on a CBEA system are memory mapped. The PPE has exclusive access to these resources using these real addresses.

In summary, the PPE is a simplified and power-efficient processor designed to work at a high clock rate. In most applications, the PPE is responsible for conducting the execution flow among the SPEs.

4.6 Synergistic Processing Elements

On its own, the PPE is no match for modern processors. By the philosophy of the design, the PPE is only there to offload and coordinate all compute intensive tasks on the remaining processing elements known as the Synergistic Processing Elements (SPEs). Similar to the design of the PPE, the SPEs are meant to be simple, power-efficient, and fast. The design and inclusion of the SPEs is meant to fill the void between general purpose processors, which are meant to achieve good performance on a wide range of applications, and special-purpose processors, which are optimized for a specific task (such as graphics processing units).

The eight SPEs that are available on the Cell processor in the PS3 are based on a novel SIMD architecture tweaked for a high throughput of mainly floating point operations. Each SPE consists of the Synergistic Processing Unit (SPU) core, a Local Store (LS), and Memory Flow Controller (MFC).

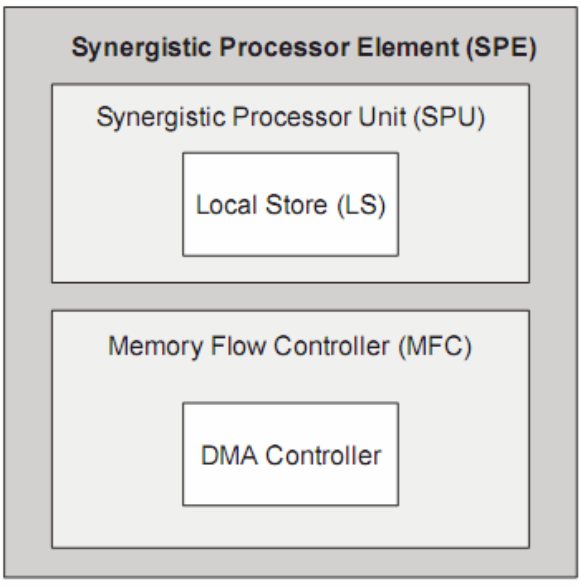


Figure 4-3: SPE Block Diagram (source: Cell Broadband Engine Programming Handbook v1.1 [51])

The floating point unit on the SPEs is heavily optimized for speed and pipeline length. In this regard, as were previous designs for the Playstation 2, the SPE processor's single point floating point calculations are not fully compliant with the IEEE754 specification (see section 4.7) which may detract researchers from using the processor for critical scientific simulations. However, these decisions were deemed acceptable for the improvement in performance and in many cases are not detrimental to obtaining valid results in the aforementioned applications. The SPE does contain a double precision unit which is compliant to the IEEE854 specification, although it is significantly (10-fold according to IBM at the ISSCC 2005) slower than its single precision brother. Therefore, while boasting a 256 GFLOPS single-point capability, it is only capable of about 25-30 GFLOPS in double precision mode.

Similar to the PPE, the SPEs are in-order two-issue cores. In addition, the SPEs do not include hardware branch prediction logic, placing a heavy burden on the compiler.

Six execution units are divided among the *odd* and *even* pipeline on each SPE. The floating point and fixed point units reside on the even pipeline, while the permute, local store, channel, and branch units exist in the odd pipeline. The following table shows the cycle times and pipeline for each instruction type handled.

Unit	**Instructions**	**Execution Pipe**	**Unit Pipeline Depth**	**Instruction Latency**
Simple Fixed	word arithmetic, logicals, counting leading zeros, selects and compares	*Even*	2	2
Simple Fixed	word shifts and rotates	*Even*	3	4
Single Precision	multiply-accumulate	*Even*	6	6
Single Precision	integer multiply-accumulate	*Even*	7	7
Byte	pop count, absolute sum of differences, byte average, byte sum	*Even*	3	4
Permute	Quadword shifts, rotates, gathers, shuffles, reciprocal estimates	*Odd*	3	4
Load Store	Load, store	*Odd*	6	6
Channel	Channel Read/Write	*Odd*	5	6
Branch	Branches	*Odd*	3	4

Table 2: Instruction types and pipeline relationships

The novel SIMD-based ISA designed specifically for the SPUs is capable of operating on sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers, or four single precision floating point operations in one cycle. Double precision floating point operations are supported, but are not fully pipelined by the FPU and take significantly more clock cycles. An interesting feature is that the same arrays in the floating point unit are used for floating point computation and integer multiplication. In this way, integer multiplications are passed into the FP pipeline which bypasses the FP handling to perform the multiply.

Table 3 describes the 6 units in the two pipelines:

Unit	Responsibilities
Floating Point Unit (SFP)	single-precision double-precision 16-bit integer multiplies conversions byte operations
Even Fixed Point Unit (SFS)	arithmetic instructions logical instructions word SIMD shifts and rotates floating-point compares floating-point reciprocal and square root estimates
Odd Fixed Point Unit (SFX)	byte granularity shift rotate mask shuffle operations on quadwords
Control Unit (SCN)	fetching and issuing of instructions to pipelines branch instructions arbitration of access between LS and register file other control functions
Load and Store Unit (SLS)	load and store instructions hint for branch instructions DMA requests to the LS
Channel and DMA Unit (SSC)	communication data transfer control into and out of the SPE

Table 3: SPE Execution Modules

The SPE execution unit has access to a large unified register file with a total of 128 registers, each 128 bits in size. Most instructions operate on the 128 bit operands by treating them as four separate 32 bit operands. The register file contains 6 read ports and 2 write ports. With most instructions having 3 source operands and 1 destination operand, this meets the requirement for having two instructions execute per cycle. The processor makes heavy use of a forward-and-delay concept to avoid access latency of a register file access during successive dependent instructions in the pipeline.

It is easy to see how reliant such a design is on having instructions and data at hand in time for execution. A cache, no matter how efficient, introduces an unreliable and unpredictable performance element, and therefore must be abandoned. Instead, each SPE includes 256 Kb of exclusive local memory known as the Local Store (LS). The LS is a private, non-coherent address space that is separate from the system address space and holds both data and instructions. It is implemented using ECC protected arrays of single ported SRAM. Local Store access times are equivalent to that of a cache at 6 cycles per access. The programmer must manually transfer contents between the LS and main memory (as well as between LSs of different SPEs) using special commands bound for the MFC. The MFC is controlled asynchronously and runs in parallel to the SPE. It is capable of sustaining 16 outstanding commands and uses the Power Architecture page protection model for its DMA-based interface. This model implies a consistency in memory mapping across the heterogeneous devices on the chip. The result of this is that memory addresses can be interchanged without issue.

The LS is physically the largest part on the SPE cell and is implemented in four separate arrays of 64 KB each. It is interfaced via a single port to reduce chip area, making it necessary to arbitrate between DMA reads, writes, instruction fetches, loads, and stores. The LS has a narrow (128 bit) and wide (128 byte) read/write port. The wide port is used for DMA reads and writes and instruction fetching and prefetching. Highest priority is given to DMA commands, followed by loads and stores. Instruction fetching occurs at all other free cycles. There exists a special no-op instruction available to the programmer which allocates cycles for instruction fetching.

The LS is connected to the main memory bus via a 128-bit memory bus. As mentioned, the LS is not transparent and the programmer/compiler has full control of its contents. To take advantage of the two available pipelines, memory and core instructions are capable of being executed simultaneously. IBM has coined a new form of parallelism called Compute-transfer parallelism. An application thread on an SPE has two threads of control - the SPU thread and SMF thread. This feature allows the compiler/programmer to decouple data fetch and use. The operations utilized by the SPU thread tend to execute on the even pipeline, while those for the MFC execute on the odd pipeline.

These design decisions do place a larger portion of the burden onto the programmer and compiler. However, they are still deemed more flexible than those of a special-purpose processor.

4.7 Floating Point Number Representation

The Cell processor was originally intended for multimedia applications, such as real-time 3-D gaming, media streaming, and signal processing. In contrast to scientific applications, these applications do not require many of the features included in the IEEE 754 floating point standard. Exact rounding, exceptions, and de-norm number handling are not critical as they make an indiscernible difference to the eye or ear of the customer. To take advantage of this fact, the SIMD floating point unit on the SPU takes several shortcuts – thus deviating from the IEEE 754 standard. Double precision support is IEEE 754 compliant, but it almost an order of magnitude slower. Newer implementations of the CBEA architecture will feature fully pipelined double precision floating point support.

The SPU floating point implementation is capable of representing a slightly larger range of normalized numbers by utilizing the least significant bit of the exponent field for the maximum value (see Fig. 4-4). The representation of positive, nonzero numbers ranges from $S_{min} = 2^{-126}$ to $S_{max} = \left(2 - 2^{-23}\right) \cdot 2^{128}$. The corresponding numbers in the IEEE 754 standard are 2^{-126} and $\left(2 - 2^{-23}\right) \cdot 2^{127}$ in which the value $\left(2 - 2^{-23}\right)$ is one least significant bit less than 2. Results which exceed Smax are clamped to Smax with the appropriate sign; those that are smaller than Smin are set to 0 (always positive sign). Infinity values are not supported.

Number Format	Minimum Positive Magnitude (Smin)			Maximum Positive Magnitude (Smax)			Notes
Register Value	x'00800000'			x'7FFFFFFF'			
Bit Fields	Sign	8-Bit Biased Exponent	Fraction (implied [1] and 23 bits)	Sign	8-Bit Biased Exponent	Fraction (implied [1] and 23 bits)	1
	0	00000001	[1.]000...000	0	11111111	[1.]111...111	
Value in Powers of 2	+	$2^{(1-127)}$	1	+	$2^{(255-127)}$	$2 - 2^{-23}$	2
Combined Exponent and Fraction	$2^{-126} \times (+1)$			$2^{128} \times (+[2 - 2^{-23}])$			
Value of Register in Decimal	1.2×10^{-38}			6.8×10^{38}			
Notes: 1. The exponent field is biased by +127. 2. The value $2 - 2^{-23}$ is one least significant bit (LSb) less than 2.							

Figure 4-4: Single Precision Floating Point Representation (source: Cell Broadband Engine Programming Handbook v1.1 [51])

Two other deviations from the IEEE standard are (a) the omission of denormalized number support with such numbers being treated as zero and (b) the inclusion of only the *round towards zero* (truncation) rounding mode.

Double-precision floating point operations are performed on the FPU unit and do support the IEEE 754 standard, but are not fully pipelined. They are performed as two double-precision operations in 2-way SIMD fashion. The operations are performed back to back in consecutive instruction slots in the pipeline and cannot be dual issued with any other instructions. Of the 13 clock cycles required, only 7 are pipelined. In addition, no instruction can be issued for six cycles after the double precision instruction is issued [51].

4.8 Element Interconnect Bus

The PPE, SPEs, memory controller (MIC), and a bus interface controller (BIC) are all connected using an on-chip, low-latency, high-bandwidth ring bus known as the Element Interconnect Bus (EIB). Each unit is connected via its own Bus Interface Unit (BIU). The EIB consists of the data network (four rings), command network (tree), and data arbiter network (star). The ring bus runs at half the clock speed of the core clock frequency and consists of independent data and command networks.

Due to the high bandwidth capability of each unit on the network (51.2 Gb/s aggregate injection and reception bandwidth) the EIB must be fast enough to avoid being a bottleneck. A total of four 16B directional data rings are included: two clockwise, and two counterclockwise. Each unit is allowed to transfer one 16B block every bus cycle and each ring is capable of processing three concurrent transfers simultaneously so long as their paths do not overlap and the source and destination distance is no more than half the distance of the entire ring.

Access to the data network is credit based with each unit starting off with a number of credits linked to the size of the command buffer within the EIB for that unit. A credit is used on the execution of a request and returned when the request moves into a further

stage in the EIB request pipeline. The central Data arbiter, connected to the units by a star network is responsible for granting access to the data rings. While the MIC is given greater priority to minimize stalling, the SPEs and other units are given equal access by using a round-robin scheme. The decision for granting access is based on two main factors: whether the total distance of the transfer is less than half the total ring distance (two of the four channels always meet this criterion) and whether the transfer would interfere with an existing transfer. Each element is allowed to have up to 64 outstanding requests (SPEs support only 16 outstanding requests).

The single shared command network is arbitrated through the use of a tree of five fully pipelined address concentrators (ACs) which handle collision detection and prevention. Each command is propagated up the tree up to the root address concentrator AC0, which can process one command per every two bus cycles.

The BIC controller is split into two separate noncoherent interfaces (IOIF0 and IOIF1). Multiple Cell chips may be interconnected via the IOIF0 interface to form one coherent ring with the help of the Broadband Interface (BIF) protocol.

4.9 Memory Interface

The Cell Processor is sandwiched in between two Rambus interfaces - the XIO memory interface and the FlexIO host data and control bus interface.

External Rambus XDR memory connects through two XIO channels – also a Rambus design. Together with the memory interface controller (MIC) on the chip, designed by IBM, memory bandwidth is rated at 3.2 GHz/s (review) per channel, which translates to a theoretical maximum of 25.6 GB/s (assuming that the memory banks are kept active continuously by request streams of the same type and 128B request sizes). Actual memory bandwidth is also limited due to typical memory operations such as refreshing and scrubbing. Interleaved read and write requests result in an effective bandwidth of about 21 GB/s due to the need for repetitive overturning of the MIC-to-XIO bidirectional bus.

Both channels can operate on eight banks simultaneously, and can operate on a maximum of 256 MB of memory (512 MB for both channels).

The MIC contains two queues for each channel – one for reading and one for writing. It performs all arbitration control ensuring high data rates. A high priority read request is supported, and takes precedence over normal reads and writes.

The system link (FlexIO) is also designed by Rambus. The parallel interface consists of seven transmit and five receive RAMBUS Redwood Rambus ASIC Cell (RRAC) FlexIO links. Each link is 1 byte wide. The interface is capable of being clocked independently, (5 GHz on the Playstation 3). Several design challenges had to be overcome to allow for the multi-hundred Gigabit aggregate bandwidth that is demanded from the Cell processor. These include channel distortion, temperature drift, and supply noise.

The FlexIO is theoretically capable of a 35 GB/s total outbound and a 25 GB/s total inbound raw bandwidth at 5 GHz. The name flex can be attributed to the interface's capability of being divided into two logical interfaces, allocating each a portion of the pins and bandwidth. This way, it is possible to optimize the interface scheme based on the devices present external to the cell chip. The Cell processor itself reserves 4 inbound and 4 output lanes for memory coherency.

There is considerable overhead being transmitted on the I/O interface during data transmission. Data and commands are encapsulated in packets which contain information such as tags, data size, command id, flow control information, etc. This overhead can have a large impact on actual throughput performance.

4.10 Previous Work on Cell Processor

The Cell Architecture, while originally designed mainly for multimedia applications, has gained much respect in the scientific community over the course of its lifespan. It turns out that the non-IEEE compliant hardware is still applicable to many applications. While not entirely a new concept, the CBEA does hold potential to accelerate the computation of existing real world problems. In [52], Williams et. al. implemented several common scientific calculations on the Cell processor, including dense matrix multiplication, sparse matrix multiplication, stencil computations, and 1D/2D FFTs, and compared the results to those on existing hardware.

The Playstation 3, which has been used in this research, has been successfully utilized for scientific research in many projects since its launch in late 2006. The most popular example is the Folding@Home project [53] which is a distributed computing project aimed at simulating protein folding with the goal of better understanding the development of human diseases. Playstation 3 users are given the option to download a Cell optimized client and run it when not using their system for other tasks. On September 15, 2007 the release of a new client lead to the breaking of the petaFLOPS barrier which was a first for any computing system in history and was recognized by the Guinness book of records [54].

Another example that is recently getting attention is the simulation of the collision of black holes and the gravitational waves that they produce in a project named the PS3 Gravity Grid lead by Dr. Gaurav Khanna at the University of Massachusetts [55]. The tightly-coupled Beowulf cluster is composed of 16 Playstation 3 systems.

Other examples include Axion Racing's utilization of the PS3 for its stereo vision algorithms in their 2007 entry into the DARPA Urban challenge [56], Security-Assessment.com's password cracking [57], and Intrusion Detection pattern matching algorithms [58].

Chapter 5: High Performance Programming on the Cell Processor

5.1 Chapter Introduction

This chapter is intended as an introduction to the tools and recommended methods and strategies for programming high performance applications on the Cell Processor. Its inclusion was deemed fitting due to the substantial effort placed into the optimization of the Multi-Layer Perceptron and Support Vector Machine algorithms for the Cell architecture. While the topics covered are described in the context of the Cell architecture, many apply and have roots in the field of general high performance parallel as well as vector programming. In fact the practices described in this chapter can likely be applied to other similar architectures. The objective is to collect the major strategies into one text, serving as a reference.

The chapter starts with a description of the development tools that were freely available at the time of this work as well as the API and other programming facilities and intrinsics for common tasks such as inter-processor communication and data transfer. The available levels of programming are compared in terms of development effort and control granularity.

Next, programming strategies are explained in detail starting at the lowest levels. Instruction issue timings, branch minimization, vectorization, and similar topics are covered first. Discussion extends to higher level strategies, such as data management, and job distribution. Finally, high level development strategies are outlined that have been made public by several knowledgeable Cell programmers.

Finally, the type of work that is suitable for the SPEs is characterized, providing a path into the next chapters that discuss the specific MLP and SVM Cell implementation.

5.2 Support and Development Tools

It can be a daunting task to begin development on a completely new and complex architecture such as the Cell Processor. The decision to do so is even more risky if the financial future of the developer (such as a game developer in the case of the Playstation 3) is at stake. IBM, Sony, and Toshiba understood this concern and made it in their best interest to educate potential programmers as well as facilitate the actual development process. For example, although occurring after the release of the Playstation 3, Sony did hold sessions in which they educated certain companies on the topic of proper exploitation of the Cell's hardware for maximum performance.

With the Cell being an open platform, there is an abundance of information posted in the form of papers, digital books, training documents, manuals, and community forums on IBM's servers. This material, which is updated on a regular basis, details both the hardware and software side of the system. Anyone interested in writing applications for the Cell, or simply in understanding the hardware can easily obtain the technical documentation. Recommended documents are included in the reference section at the end of this chapter. In addition to supporting documentation, there are many freely available, yet quite advanced tools written specifically to help in maximizing this hidden performance. The Software Development Kit (version 3.0 at the time of this writing) that is available for download from the IBM developerWorks website is a CD image containing all that is necessary to get started given a supported Linux operating system. An optional extra CD image containing prewritten code examples, some lesser used tools, and useful SPE-targeted libraries is available as well.

5.2.1 The Full System Simulator

The most useful tool within the SDK while developing code is systemsim - the Full System Simulator. This piece of software (which runs on PowerPC, x86, and x86-64) simulates systems based on PowerPC (or related) architectures such as the Cell itself. Cell support was designed and utilized throughout, and after, the product development cycle of the actual Cell hardware chip as a means of obtaining vital feedback that is just not obtainable from actual hardware. In fact, Linux was running on the simulator two years before the actual hardware was available. Today, with the processor available on the market, the application is freely available for Cell developers who use it as a debugging and/or performance evaluation tool [59].

While the simulator is not perfect, it is considered to be fairly accurate and complete. All the elements needed to run an operating system such as Linux are simulated. In fact, when running Linux-based applications, the simulator first loads a PowerPC based Linux kernel from which the application under test is executed via the simulated command shell. It is also possible to run in standalone mode, in which no underlying Linux OS is utilized. The usefulness of the Full System Simulator lies in its ability to expose the programmer to every essential component of the chip at any cycle in the simulated program's execution. Simulated components include all the elements pictured in Fig. 5-1. One notable difference from the actual hardware is the simulation of DDR2 memory instead of RAMBUS. The Instruction Set Architecture is also modeled down to the specifications.

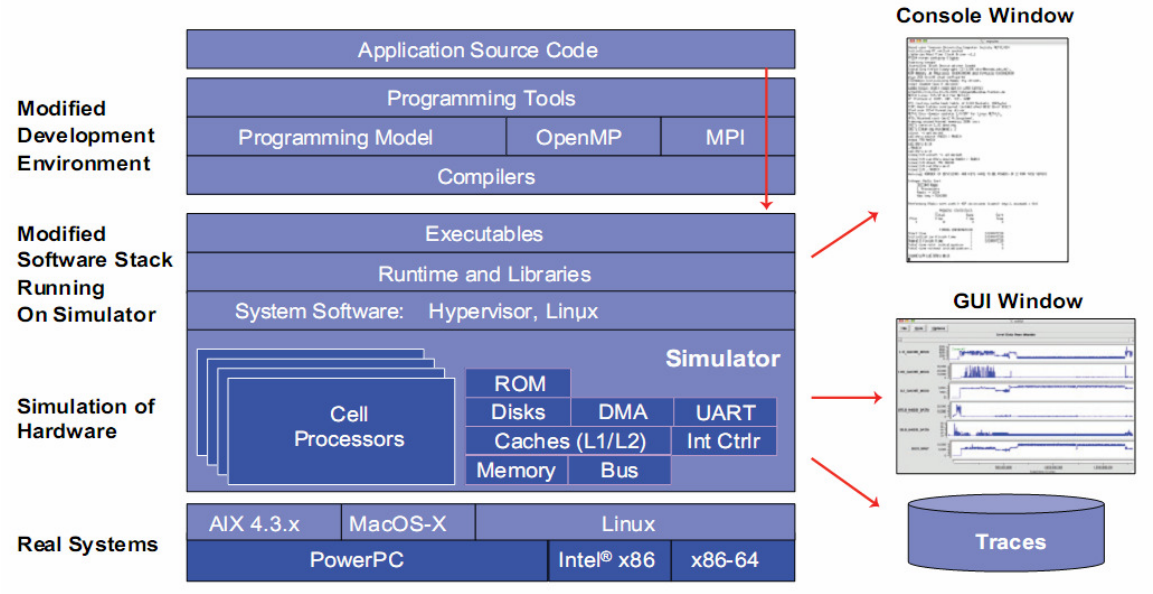


Figure 5-1: The layers of the Cell simulator (source: CBE Tutorial v2.1 [60])

Upon starting the application, three windows are presented, as shown in Fig. 5-1: a command line and graphical interface into the simulator and a text console as it would appear on the screen of the simulated system. The graphical simulator interface provides a display of the state of the simulated system, including the PPEs and SPEs. It allows for the viewing and modification of various items such as memory, register, and channel contents through dialogs and the graphical representation of system state, history, and statistics.

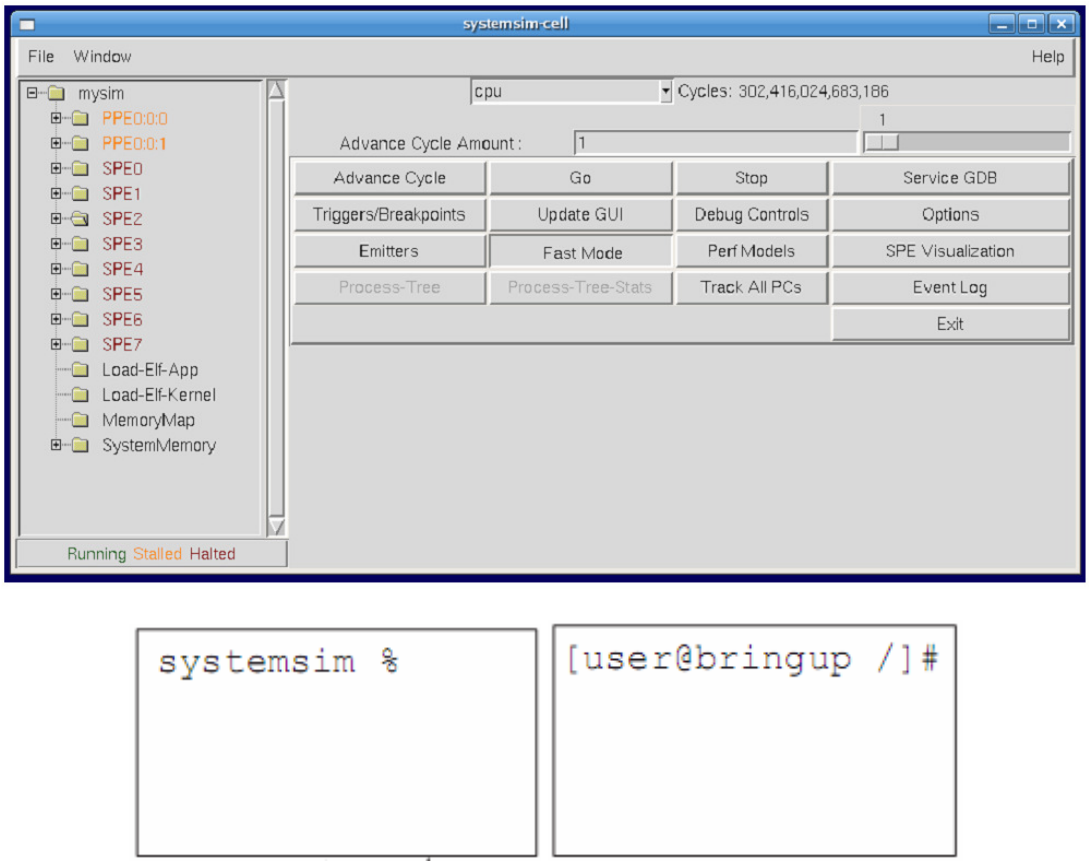


Figure 5-2: The UI of the cell simulator (source: CBE Tutorial v2.1 [60])

The most useful feature of the simulator, which has been utilized several times in this work, is the ability to collect a summary of per SPE performance statistics for any portion of the code. The programmer uses three function calls to utilize this facility: *prof_clear*, *prof_start*, and *prof_end*. The actual code generated has no performance effect when running on real hardware and is utilized only by the simulator environment. *Prof_clear* zeros all collected statistics for that particular SPE. The simulator begins analyzing and collecting simulated events on *prof_start* and ends at *prof_end*. An example of the resulting report, which is updated in real time during program execution is shown in Fig. 5-3.

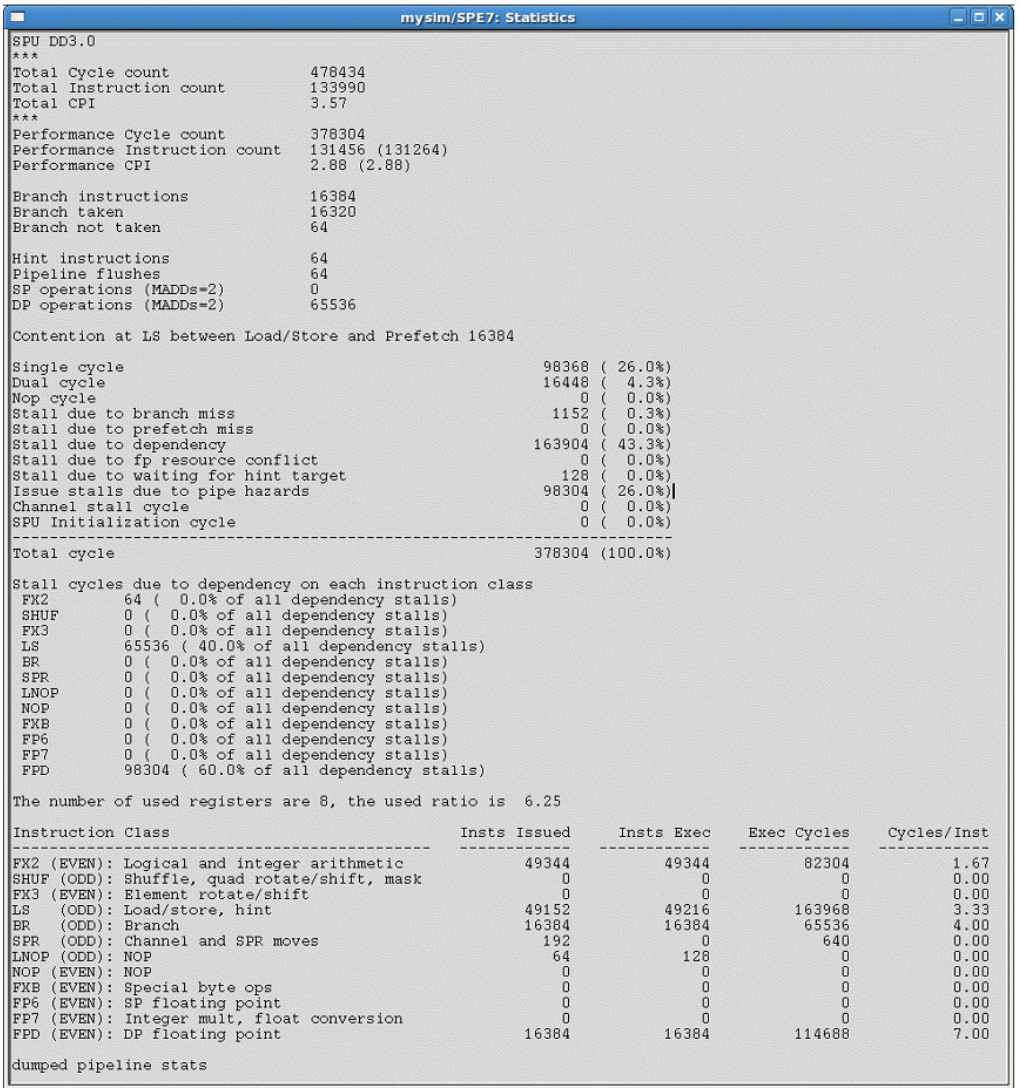


Figure 5-3: Profiling results (source: SystemSim Users Guide [[])

The information presented in the profiling statistics window gives a good representation of how well the code is utilizing the available hardware. As evident in the example, only 4% of the cycles utilized dual issue, and nearly half the cycles were spent waiting for a data dependency resolution. Looking at the number of DPFP instructions executed, and the small number of registers used, it is likely that the code performs a simple DPFP operation in a loop. DPFP operations are not well pipelined on the SPEs and are likely the source of the dependency stalls.

The simulator has access to the host system via two interfaces – the *callthru* utility which allows for the transfer of files and the *BogusNet* interface which sets up a virtual network interface with the host (*BogusNet* is actually an extension of *callthru*). By enabling the *BogusNet* interface, the possibility for remote debugging of simulated applications via the GDB (Gnu Project Debugger) debugging toolset is opened. The user starts the program

under test (PUT) under the context of gdb-server in the simulator and then attaches to the server using a GDB client on the host computer over *BogusNet*.

The GDB toolset which is included in the Cell SDK has been enhanced with some useful Cell-based functionality. The user is capable of viewing and stepping through the source code for both PPE and SPE modules (separate PPE and SPE executables are usually necessary as will be mentioned next). The enhanced debugger has the ability to examine events, signals, mailbox contents, and DMA transfers via new *info spu [item]* commands as well as detect bus errors on DMA transfers. The common functionality such as setting of breakpoints and viewing and modification of raw memory is also included.

The simulator includes a Tcl interpreter which allows for the creation of scripts that can be programmed to trigger on a certain simulator event. These scripts may be used to control the simulator via the available simulator commands. For example, profiling events can be bound to a script which captures the statistics and files them for later processing. This allows defining multiple profiling sections within the code, each filed separately.

The *Emitter* Framework is a facility for decoupling the production and processing of simulator events. *Emitter* readers can be written and attached to buffers associated with events for the purpose of graphing or collection event statistics. Examples are included in the SDK CDs. Further information is included in the SystemSim Users Guide.

Two other useful tools available on the SDK are Asmvis (Assembly Visualizer for Cell Broadband Engine) and FDPR (Feedback Directed Program Restructuring) Pro. The first is a static performance-tuning utility. This tool allows the programmer to manuall open a generated assembly file, navigate, and reorder instructions by hand using an easy to use GUI interface. The tool's built in ruleset prevents the user from accidentally breaking the dependency logic. This tool can be used to optimize specific sections in the code that are known to be executed often [61]. FDPR Pro is a dynamic performance-tuning utility. This utility optimizes the execution of a program image by collecting runtime information under a typical workload. Once the information is collected, the image is restructured to optimize performance [62].

5.3 CBE Embedded SPE Object Format

When writing software for the Cell Processor and using a dual-source compiler (such as the GNU GCC or IBM's dual-source XLC compiler), the PPE and SPE binaries are written and compiled separately using different compilers, each specialized for one of the two instruction sets. The traditional tool chain infrastructure and Executable and Linking Format (ELF) that is defined in the Tool Interface Standard does not support inter-architectural linking and therefore makes it difficult to define inter-architectural symbols and bindings. The CBE Embedded SPE Object Format (CESOF) was developed for this reason and allows for the embedding of the SPE executable within the PPE executable. The CESOF format is an application and extension of the ELF standard. It does not modify it in any way. As another option, it is possible to load the SPE executable at runtime using the underlying operating system's file operations, thus keeping the two binaries separate. In addition to the two methods above, there is also a runtime

environment for running standalone SPE programs, known as *SPUlets*, by simply executing them on the command line. For more information, see [[].

It is important to note that shared data structures may be represented differently on the two architectures and special care must be taken when including the same header files between source code targeted for the two architectures. For example, when compiled to make use of the PPE's 64-bit architecture, pointers on the PPU are 64 bits long, while on the SPE they are always 32 bits long. In this situation, it is advisable to use a custom address data type such as shown in Listing 10.

```
typedef union
{
  unsigned long long ull;
  unsigned int ui[2];
}
addr64;
```

Listing 10: Shared data structure of a pointer type

It allows the SPE to receive effective addresses via any of the communication mechanisms available between the PPU and SPE.

5.4 Levels of Programming

On the SPEs, the programmer has the option to write code on one of several levels, including hand crafting assembly code, as well as C/C++ code utilizing some of the available intrinsics for SIMD processing and MFC interfacing. The compilers available, while generally very capable, are unable to vectorize scalar code and can only exploit optimization opportunities that are given to them by the programmer. It is, therefore, a big task for the programmer to write code in a manner that the compiler can recognize. Naturally, programming methods may be combined so that "hot" code is hand optimized for best performance. Compiler generated assembly code can also be examined and tweaked, although this is generally very difficult unless the developer has a deep understanding of the chosen compiler's methods.

The following section exposes several design strategies that can have a huge impact on execution time and hardware utilization. Tips are also provided to ease development and testing on this complex architecture. The concepts are presented in order from low level to high level. In general, the hardware has a greater influence on code at the low level, and the algorithm has a greater influence at the higher levels. In practice, the tactics described in these sections would generally be applied in reverse order, starting with a high level code/algorithm organization, and optimizing segments of code as time goes on. The reverse approach in this document makes more sense when introducing the ideas.

5.4.1 Low Level

As expected, the low level design choices are made almost exclusively due to the underlying hardware. On the Cell, it tends to be the case that faster versions of code are more complex and are more prone to programmer mistakes (and frustration). As a good

practice, it is useful to write non-optimized and easy to understand code sections first as a baseline. Once verified, new and optimized sections can be appended and surrounded by conditional compiler directives so that the multiple versions can be toggled for debugging. Retaining the non-optimized sections in the source code also makes it easier to understand the purpose of code sections written previously or by someone else. When experimenting with these optimizations, it is useful to employ systemsim and utilize the selective profiling feature (*profile_start*, *profile_stop*) to study the improvements.

5.4.1.1 SIMD

One of the major factors to consider when writing software for the SPEs is that, due to the 128x128b unified register file and SIMD nature of the function units, they can load and store only 128b at a time (known as *quadwords*, or *qwords* for short). With the *qword* being the only native data type on the architecture, reading a scalar value of any size in the Local Store (stack or global space) requires the loading of that value as well as the spatial overhead that surrounds it. In addition to this overhead, for the functional units to process a scalar value, it needs to be loaded into a preferred slot within that 128b register. For this to occur, the compiler (given low optimization options) generates code that rotates the 128b register so that the value is placed in the correct slot. Writing the variable requires reading the content of the destination memory into a register, inserting the scalar value, and writing back the result. Higher compiler optimization options align the scalar variables so that the value is already in the proper slot. In all cases, however, the LS space is not utilized efficiently making it a priority to minimize the use of scalar code whenever possible. In cases where they are necessary, the __attribute__((aligned(128)) directive should be used to help the compiler align the variable on a 128b boundary, placing it into the proper slot.

Utilizing the SIMD hardware on the SPEs is one of the strongest methods for exposing parallelism on the entire Cell architecture. To encourage its use, C/C++ language extensions are provided which define vector datatypes and so called intrinsics. All vector types are 128b in length and are always aligned on 128 bits (fitting the hardware model) thus eliminating the need for shuffling of data when loaded from memory. Intrinsics are functions which provide all the necessary operations for the processing of vector types. They include floating point operations, bit operations, comparisons, *quadword* shifts and shuffles, element-wise rotates, conversion between float and scalar, etc. Two levels of intrinsics are provided: composite and specific. The composite intrinsics are easier to use as they automatically select the proper assembly instructions based on the data types, and may generate multiple assembly instructions depending on the context. The programmer does not need to remember every version of the instruction. Composite intrinsics have the form spu_*instruction.* A specific intrinsic, on the other hand, is mapped to exactly one assembly instruction and is called using the si_*assemblyinstruction* form. In both cases, the programmer has more direct control over the instruction code that the compiler generates. For example, the programmer can use the shuffling and rotating intrinsics to override (and possibly optimize) any data alignment that the compiler would otherwise generate.

5.4.1.2 Branch Reduction

Due to the hardware's lack of dynamic branch prediction, there are numerous strategies that can be utilized to improve performance. Often, the value of a variable is determined based on some condition on the value or values of other variables. Listing 11 summarizes the idea.

```
if( test_var meets some condition )
   determine value_a
   dest_var = value_a
else
   determine value_b
   dest_var = value_b
```

Listing 11: Generic branch structure

The SPE instruction set features two separate instructions – vector comparison and select – that can be utilized in tandem to obtain the same result. The method often takes fewer cycles than it would to resolve a branch condition especially if making use of the entire set of elements within the vector. The compare and select instructions are exposed using intrinsics as discussed in the previous section. Sticking to the same variable names as above, the procedure utilizing these intrinsics is shown in Listing 12.

```
Determine value_a
Determine value_b
select_vector = spu_comparison( test_var, compare_value )
dest_var      = spu_sel( value_a, value_b, select_vector )
```

Listing 12: Alternative without conditional control flow

As the general example shows, both the code required to calculate value_a and value_b need to be executed but only one of the values is actually kept. The decision to use this method often comes down to the relative computational requirement for computing both values and to resolve a branch condition (and performing one of the computations).

In some cases, it is known that the condition evaluated will be either true or false for a majority of the time. The instruction set provides special branch hint instructions that, if strategically placed, can be used to explicitly direct program execution. If the hint is incorrect, the pipeline ends up being flushed and a stall occurs as new instructions are loaded from the correct location. In order to be effective, the branch hint instruction needs to appear several cycles before the branch resolution instruction.

5.4.1.3 Dual Issue

As covered in Chapter 4, the SPEs have two instruction pipelines and allow for the dispatching of two instructions on one cycle. While the compiler does a good job at aligning instructions in the proper locations by inserting NOPS where necessary, the programmer needs to choose instructions carefully to satisfy the remaining conditions that need to be met for dual issue to occur. In many situations, it is often possible to perform the same function using different sequences of instructions. The programmer

should be aware which instructions execute on which pipeline and attempt to distribute them evenly between the two.

Loop unrolling – a common practice in code optimization – is an excellent way to increase dual issue rates. Doing so reduces the dependency chain between instructions giving the compiler more instructions to shuffle around. The technique also reduces branches, further improving performance. In many cases, when the number of loop iterations is unknown, extra effort must be put into writing the preamble and/or epilogue. When the number of iterations is small and known, the loop can be removed altogether. The main disadvantage of loop unrolling is an increased code size – something the SPE gives the programmer a limited budget of. It is up to the programmer to determine those loops which are executed most frequently and with the most iterations and give those higher priority.

5.4.2 Mid Level

The optimization of code sections can only help as long as there is a smooth and constant flow of work in the form of commands and/or associated data. Fast code is only fast when it is executing. By analyzing the actual implementing algorithm, mid level optimization can be exercised to keep the processors busy and keep them from stalling. This section focuses on that very task.

5.4.2.1 Programming the DMA Controller

Each SPE contains its own dedicated Memory Flow Controller. This module is primarily responsible for the transfer of data between the given SPE's LS and all system-wide addressable hardware. The SPE performs communication with its MFC via so called channels. The C/C++ SPE Extensions include MFC functions which automatically generate sequences of channel commands that perform common tasks such as queuing a DMA transfer request or checking on the status of an active request. The most basic command is the request for the start of a DMA transfer. Parameters include the system-wide effective address of the source, the local destination address within the LS of the current SPE, the size of the transfer in bytes, and the tag for the DMA request. The source and destination addresses need to be 16 byte aligned (128 byte aligned for optimum performance), and the size of the transfer needs to be a multiple of 16 bytes (again, 128 bytes is faster). The tag parameter does not need to be unique and is used as a means for subsequent inquires into the status of the transfer(s). It can also be used as a way to enforce transfer order as will be explained in the next paragraph. Two additional parameters – the transfer class ID (*tid*) and replacement class ID (*rid*) – are available. They may vary between CBE implementations. On the Cell, the *rid* influences L2-cache and TLB (translation lookaside buffer) replacement. The *tid* influences the allocation of bus bandwidth. These two parameters were not used in this work. More information can be found in [51].

By default, there is no guarantee of the order in which the DMA executes the transfer requests. The DMA controller has its own techniques for minimizing total transfer time and may choose requests in any order it deems fit. To force the order of transfer request, two command modifiers are available: fence and barrier. By including the fence modifier,

the controller is forced to wait until all previous transfers with the same tag are complete before continuing with the current one. A barrier is a stricter version of a fence, in that the current transfer is forced to complete before any subsequent transfers with the same tag.

The asynchronous nature of the MFC in relation to the SPE allows the programmer to efficiently parallelize in-flight DMA transfers and data processing and in effect hide or minimize stalls that result from a lack of input data. This technique is known as multi-buffering, and is described next.

5.4.2.2 Multi Buffering

A traditional method for hiding memory access latencies in similar architectures (such as graphics chipsets) is known as double, triple, or multi buffering. On the Cell, implementing double-buffering requires the reservation/allocation of two buffers on the LS, preferably of equal size. Both, the MFC and SPE have direct access to the LS and can read and write from it the same time. The concept is to always have the SPE working on one buffer, while the DMA controller is sending or receiving contents into the other buffer. The idea is expressed in Fig. 5-4. Essentially, the buffers make up a circular queue of size 2. By extending the number of buffers the concept generalizes to multi buffering. The SPE always works on one buffer at a time and, optimally, will have the succeeding buffer filled and ready by the time it is done with the current one.

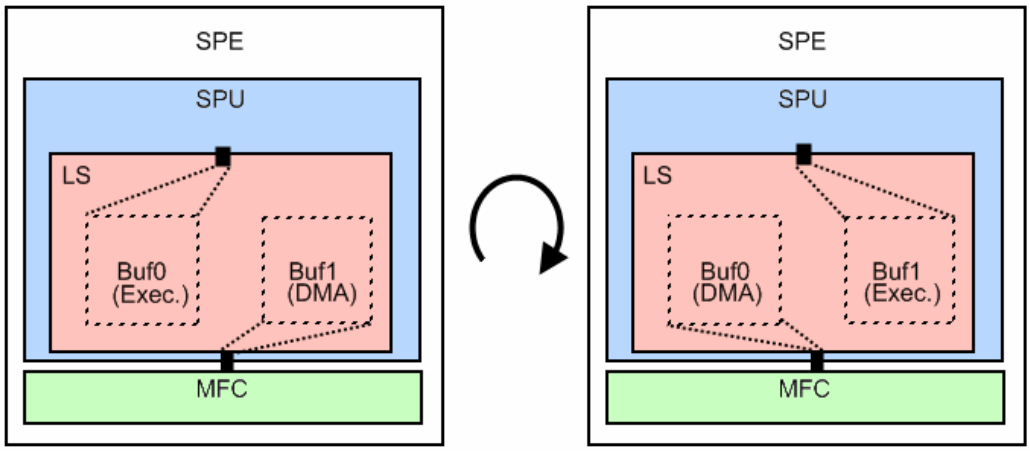


Figure 5-4: Double buffering

The main drawback to multi-buffering is the need for additional memory for the extra buffers. Given that there is M memory available and n buffers, the size of each buffer becomes floor(M/n). In the case of the SPEs, the buffers need to be aligned on a 16b boundary and be of a multiple of 16 bytes each introducing some padding overhead. Although usually insignificant, the additional multi-buffering code and status variables add to the overhead as well.

5.4.2.3 Inter-SPE communication

Each SPE's MFC is capable of reading from and writing to all of system-wide memory. During the initialization of the SPEs, the programmer has the option to map the LS's and so called *problem state* of each SPE to this globally accessible address space. The problem state area is takes the form of a c struct and includes the addresses of various SPE registers related to multi-source synchronization, proxy DMAs, mailboxes, and signal notifiers. The problem state area and LS base addresses can be obtained by using functions in the SPE library on the SPE context objects during SPE initialization. Once the SPEs are running, a collection of this data can be sent and stored in a local table on each SPE. Having this information, any SPE can DMA contents into the LS of any other SPE as well as communicate with the other SPEs via the memory mapped mailbox and signal registers taken from the problem state area. Giving each SPE this much power gives the developer much flexibility in the communication model. It is up to the programmer to develop a communication model that fits well into the application.

5.4.2.4 SPE Shaders

A flexible, but potentially difficult, programming technique is to design small, dynamic, programs which are automatically sent via DMA transfers onto the SPE whenever necessary. These dynamic code fragments are made to be self contained and are transferred onto the SPEs in the same manner as normal data. In fact, code can be handled in the same way that data is on the Cell. Multi-buffering on code fragments, for example, is not out of the question. Insomniac games have shared their method for doing so in the context of the Cell processor [1].

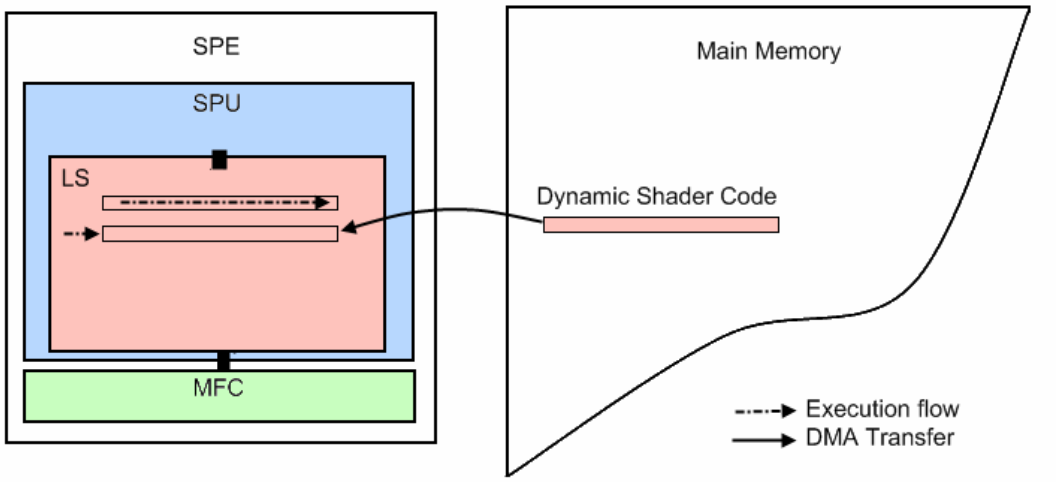


Figure 5-5: SPE Shader concept

The idea for using dynamic code fragments was inspired by the shader programming model that is utilized when programming graphics cards. In fact, Insomniac refers to their system as SPE shaders. A shader is a specialized fragment of code used in an existing system that modifies the system data using a custom input/output interface. The location of the fragments is predetermined by the system they are part of. Logically, the fragments may represent asynchronous data processing utilities or be part of a system pipeline, etc.

The usefulness of this technique on the Cell is apparent when the application implementation requires more storage than the SPEs' LS can provide as well as at times when specific code sections are needed only sparsely. The method, once designed, is easy to conceptualize and maintain. Also, being self-maintained and treated as data, the shader management library utilizes the same techniques as those for data transfer.

The Cell development library has support for the programming of so called code overlays. It is a method, similar to the one described above, in which designated code is not loaded into the LS until it is necessary. Code is divided into segments. When one segment calls another segment, a table in memory is indexed to see if it is in memory. If not, the segment is transferred in, possibly overwriting another segment, before execution continues. For an introduction to overlays on the Cell see [51]. It was Insomniac's opinion that this model was too complex and had unnecessary overhead for their purposes, leading them to come up with their own version.

The difficulty with developing a custom implementation of this technique is implementing the library for managing these fragments. It can be very time consuming to develop a library that is both: efficient and flexible. Also, as is evident from the above, the final product may work well for some applications, but not others.

5.4.2.5 Job Distribution and Synchronization

Besides multi-buffering, there are other implementation decisions that can influence the utilization ratio of the SPEs. These decisions are largely influenced by the application being implemented. The following are common scenarios and good practices given the most widely used server programming model in which each SPE receives one or multiple jobs from the PPE to process.

When writing parallel code, one of the first tasks is to find a method for dividing the overall work into a number of tasks, or jobs. In the best case scenario, this number is known ahead of time, the jobs are not dependent on one another, and each job takes the same amount of time to complete. A fast implementation would involve sending each SPE an equally sized list of the jobs they are responsible for. Each job list element should include information about the effective addresses of any inputs, the effective addresses of output destinations for any produced data, and any other required job related information. Another, slightly better, method would be to have the SPEs all receive the same exact job information and a personal SPE ID. This way, each SPE would select and transfer its jobs as a function of its ID. These methods require minimal communication between the PPE and SPEs during the job processing step and if proper multi-buffering is used, make it easy to obtain close to optimal performance.

The first complication may arise due to uneven or unpredictable job completion times. An effective solution may be to create a separate job buffer for each SPE that the PPE checks and fills periodically from the main job buffer. Completed jobs should be marked as such in the SPE's job buffer in main memory so that the PPE can replace it. While it is possible for the SPEs to arbitrate on a common buffer by performing atomic operations, this would require far more synchronization. A simpler solution is to send exclusive batches of jobs to all SPEs. The number of jobs should be large at the beginning, but

decrease in size as there are fewer jobs available in the global job pool. Smaller sized batches imply more communication overhead at the expense of better job distribution.

Another complication may be the interdependence between different jobs. If the dependencies are known, the PPE can dispatch batches of independent jobs at a time. Otherwise, if the dependencies are dynamic, the PPE can use, for example, a dependency tree and evenly queue up those jobs which are ready to run among the job buffers previously described. If there is too much dependence, however, at least two options exist. The first option involves finding parallelism within the jobs themselves and distributing that between the processors. The second option is to forgo the use of the data distribution model for something else, such as the pipeline programming model described later.

A global strategy that applies to all these methods is to keep the large portion of the input and output data out of the PPE's L2 cache. To promote this, the PPE should not be accessing the data until the SPEs are finished. DMA transfers between the LS and system memory is characterized as having high bandwidth and moderate latency; those between the LS and the PPE's L2 cache are characterized as having moderate bandwidth and low latency.

5.4.2.6 SPE-Initiated DMA transfers

Each SPE's DMA controller has a proxy DMA queue that functions similarly to the main DMA transfer request queue with the exception that it may be accessed from other processing elements. The PPE, for example, is able to queue up DMA transfers between main memory and the local SPE's LS. While attractive at first, the use of this queue is not recommended if high performance is desired. Rather, the SPEs should perform their own DMA transfers for several reasons: there are more of them, the proxy queue is half as deep as the main queue, it is easier to verify DMA completion when pulling data on the consumer, and the number of cycles to queue a request locally is smaller.

5.4.3 High Level

In this section, issues surrounding the decisions for programming models, the structuring of the algorithm itself, distribution of workload, and development and testing strategies are laid out. Many of the tips have been shared by Insomniac at the Game Developer Conference 2008 in San Francisco.

5.4.3.1 Data Design over Code Design

When designing high performance code on the Cell it quickly becomes evident how important data management is. Designing for data is just as, if not more important than code design. For example, an algorithm may have a multi-stage data processing step that needs to execute on small chunks of data. The traditional, synchronous, model in which each chunk is processed individually would be very inefficient if implemented on the Cell. First, the processing of each chunk on an SPE includes the time needed for synchronization between the PPE and SPE. Secondly, such a model is inherently scalar, causing the underutilization of the SIMD functional units, with many dependency chains, causing dependency stalls. The SPE local store would most likely not be utilized to the

full extent that it could be, either. The final code would most likely not scale well to larger data sets and have poor data locality, hence poor cache utilization.

The proper procedure would be to group, or compress the data together, while possibly breaking down the multi-stage data processing step into shaders or overlays for example. Grouping data together reduces synchronization requirements and introduces opportunities for instruction level parallelism via SIMD utilization, loop unrolling, dual-issue, etc.

Having explicit control over DMA transfers puts the burden of managing what the code has access to at any point on the programmer. This may be irritating, although having control over what is placed into the LS (which is technically a user-controlled cache) encourages the grouping of data into greater chunks and results in excellent data locality. The challenge is to place much more emphasis on data management rather than only on execution flow.

5.4.3.2 Minimize Synchronization

When laying out the data and program flow, there will inevitably be positions at which synchronization between processing elements is necessary. Due to the expensive overhead, synchronization should be minimized whenever possible. One way, for example, may be to combine multiple synchronization points into one by combining independent processing steps into the same block.

5.4.3.3 Maximize SPE Usage

The SPEs are designed to be very fast data processors – in most cases faster than the PPE. It is therefore in the best interest of the designer to keep the SPEs doing as much productive work as possible. In some cases, it may be acceptable to offload even scalar code that has many branches. The PPE's main task should be to simply shuffle things around – acting as a SPE controller. It is best to think of the SPEs as streamed data processors. With the help of an SPE shader-like model and well defined data flow, many algorithms can be structured to function well using this model.

5.4.3.4 Pipeline Programming Model

The server programming model has been described already in the previous sections and will not be repeated here. Another programming model, which may be advantageous in rare situations, is the pipeline programming model. Given the flexibility that the hardware exposes in terms of creating communication models, the SPEs can be logically arranged in a chainlike fashion such that the PPE has access to the two ends (for input and output). Each SPE in the chain, or pipeline, is responsible for a sub portion of the entire process. It provides its output to the next processing element (PE) in the pipeline and obtains inputs from the previous PE at every time step. The main difficulty, and reason for its sparse usage, is not only the added communication complexity, but the difficulty in splitting a process into a predefined number of equally work-intensive and deterministic sub-processes. For this reason, the data distribution model is much more popular.

5.4.3.5 Reverse Pipeline Design

The pipeline model may be more common in gaming applications, however. A main point driven by Insomniac [63] is the importance of designing multi-stage algorithms in reverse order. Using a transformation pipeline for glass physics as an example, they discussed the inter-stage interfacing issues that appeared only after certain parts were already completed. Their argument for this design process is that it forces the programmer to think about earlier portions of the pipeline in advance and prevents the "code as you design" syndrome. It is easier to fix problems in the front of final code as opposed to wildly patching code all over the place. As stages are designed, they are tested by submitting dummy inputs and checking for valid outputs.

5.4.4 Generality vs. Performance

In developing applications, code generality is highly regarded. A software company is often defined by the libraries of existing code that it can reuse in its current and future products. Developing highly tuned and optimized applications, especially on complex hardware such as the Cell, however has different priorities. For best performance, code is highly tailored to the data flow stemming from the algorithm. Generality of Cell software can be increased by writing helpful libraries, such as Insomniac's SPE shader library. A programmer may also develop low level chunks, or modules, of code to perform common operations, such as a matrix multiplication module, or binary search module. The potential for code reuse does exist, but may require extra thought.

5.5 Programming the PPE

Programming the PPE for performance is not as critical as it is in the case of the SPEs, although there are some strategies that may be followed. In some cases, it may be beneficial to accelerate PPE processing so that it can keep up with the SPEs and be ready to provide new tasks or data. In other cases, the PPE may be individually programmed to act as another major processing element in the application.

5.5.1 Altivec

Because it is based on the PowerPC 970 architecture, the PPE has built in SIMD support in the form of the Altivec instruction set. This allows the PPE to perform similar SIMD operations as are performed on the SPEs. Likewise, similar alignment constraints apply. The Altivec instruction set is accessed using similar, but differently named, intrinsic functions and contains an equivalent for most of the SPE intrinsics and more. In addition, the instruction set contains predicates which begin with *vec_all_* or *vec_any_*. The instructions in this class operate on the entire vector in an AND or OR fashion and return a scalar value.

5.5.2 Multi-threading

Being a two-way multithreaded architecture, the PPE can run two threads simultaneously with the condition that the two threads don't access a shared resource at the same time. Multithreading is, therefore, recommended when at least one of the threads experiences heavy stalls due to cache misses or instruction dependencies.

5.5.3 Self-managed cache

If data access is predictable, the programmer has the option to manually pre-fetch cache blocks using special *dcbt* (data cache block touch) instructions. Two versions of the instruction exists: classic and enhanced. The classic versions allows loading of data into L1 cache and enhanced allows loading of data into L2.

Chapter 6: Implementation of the Multi-Layer Perceptron on the Cell Processor

6.1 Chapter Introduction

In this chapter, and those following, the focus will shift to the actual implementation. The current chapter describes the implementation of the Multi-Layer Perceptron (MLP). Starting out with a top-level description of the programming model and decisions, the focus moves into a detailed description of each step along with reasoning for the decisions made. Due to efforts placed into optimization, design generality was reduced in some areas, and thus, it may be necessary to reference ideas and design details across sections within the chapter. It is recommended that the chapter be read in order.

For a detailed description and brief history of the MLP algorithm, see Chapter 2. As a review, the algorithm consists of two tasks that are repeated until convergence. Both tasks – forward and back propagation – are attractive candidates for parallelization due to the heavy use of matrix-vector multiplication. In the standard MLP architecture, each neuron within a layer (hidden and output) is connected to all neurons on the previous layer. Each of these connections is characterized by a weight (usually a floating point representation in software). As introduced in chapter 2, the forward propagation step from the point of one neuron is taken by performing the following calculation:

$$x_n^j = F\left(\sum_{l=0}^{L_{n-1}+1}\left(w_n^{l,j} * x_{n-1}^l\right)\right) \tag{6.1}$$

It is easy to see how this operation can be expanded to multiple neurons on a layer by introducing a matrix vector operation, as so:

$$F_v\left(A\bar{\mathbf{x}}\right) = \bar{\mathbf{y}} \tag{6.2}$$

The matrix A is a *weight matrix* in which each row represents the weights connected to exactly one receiving neuron. The vector x embodies the output values from all the neurons on the previous layer (input or hidden). The function F_v is a vector version of function F.

Backpropagation may also be represented as a matrix vector product. As introduced in Chapter 2, the following equation is used to calculate the delta values for each neuron:

(6.3) $$\delta_j^l = \frac{\partial F\left(z_j^l\right)}{\partial z_j^l} \sum_{k=0}^{L_{l+1}} \delta_k^{l+1} w_{jk}$$

The equation is easily represented as a matrix multiplication:

(6.4) $$\overline{\delta}^l = \frac{\partial F_v\left(\overline{z}^l\right)}{\partial \overline{z}^l} A^T \overline{\delta}^{l+1}$$

The problem with this approach, as shown, is that the Backpropagation step requires a transpose of the weight matrix. When performing batch learning, a good design strategy is to store a separate copy of the matrix in its transpose form. Because values within the matrix are modified only at the end of each epoch, random matrix accesses are infrequent, and do not introduce a large computational penalty. This approach is not used in this work. Instead, a column-wise matrix vector product algorithm was devised. It will be explained in this chapter.

The MLP software in this work supports both fully-connected and convolution layers. While the fully-connected layer implementation makes use of the matrix-vector multiplication, exposing parallelization potential, the convolution layer implementation cannot utilize this approach and required more algorithmic analysis. Details of the final implementation follow in the corresponding sections.

In the following chapters, any reference to the next (or right) layer refers to the layer closer to the output, and vice versa.

6.2 High Level Implementation Overview

The MLP code was intended to be used as a library, and thus can be broken down into well defined steps that are taken during initialization of the network and the main training forward/backward propagation loop. In this section, a summary of the steps are given. Both, the fully-connected and convolution layers have similar APIs and are treated in a similar way. The internal operations and algorithms differ, however, and are described in much more detail in the sections that follow.

6.2.1 Programming Model

The high-level programming model used in the implementation is the function offload model utilizing data parallelism. Layers are processed synchronously as data propagates forward and backward through the MLP. The computation required at each layer is distributed among the SPEs. All SPEs contain the same code, containing all the functionality needed for the operations required of them. The SPE program consists of a main outer loop in which it waits for a command from the PPU (via a mailbox channel). The incoming command specifies the operation and parameters. The 32 bit command consists of 8 bits of the command identifier, and 24 bits of parameters, in that order. On completion of the operation requested, the SPE signals the PPE and blocks waiting for additional commands.

All SPE-unique memory addresses and job information is uploaded to the SPEs during layer initialization. Each SPE, thus, holds a table of personalized parameters. Throughout execution, this table is indexed for these parameters, abandoning the need for additional communication of parameters. In effect, each SPE has the same code, but is initialized with unique and tailored information. Common synchronization points (barriers) occur at the completion of each layer by having the PPE wait until all SPEs reply with a mailbox message. This model is simple to implement and applicable to the problem due to the static and deterministic computation times of assigned workloads.

6.2.2 Implementation Overview

In the following steps, unless otherwise noted, the operation occurs on the PPE.

6.2.2.1 Initialize SPEs

A single SPE binary is uploaded to all the SPEs. The SPE software contexts are created and the SPEs themselves begin executing the binary. At this point, each SPE halts, waiting for a command from the PPE.

6.2.2.2 Create Layers and Set up Values

At this point, every layer that makes up the full network is created individually on the PPE. Layers may be created, or loaded from a file.

6.2.2.3 Convert each Layer into Optimized SPE Layers

Each created layer is converted into a special format optimized for use by the SPEs. Each optimized layer allocates its own memory that is padded and memory aligned per DMA requirements.

6.2.2.4 Connect SPE Layers

An API function is called taking two layers as parameters. Internally, the function completes the setup of any parameters of each layer. This includes, for example, the source and destination addresses for input and output data. It is at this point that much of the neuron and weight memory is allocated.

6.2.2.5 Divide workload among SPEs

With all required information present, the workload at each layer is distributed algorithmically in an effort to balance computation load and reduce any PPU involvement if possible. The product of this step is a list of children jobs belonging to that layer. These jobs are assigned evenly across the active SPEs. Any job distribution related layer parameters are also filled in.

6.2.2.6 Initialize SPEs with Personalized Data

Because the job distribution for each layer does not change throughout execution, each SPE is given information about the location of their jobs in main memory on a per-layer basis. Each SPE holds a table within the LS in which it stores this information. Each layer is assigned a unique ID (starting at 0). This ID is used by the SPEs to index their tables.

Thus, only the ID, type of layer, and operation (forward of backward propagation) needs to be transferred to the SPE at each step.

6.2.2.7 Start Learning Loop

The top-level loop consists of executing the forward propagation and back propagation functions for each layer in the right order. For any given layer, each SPE is given a command specifying the requested task and id of the current layer. The SPEs reference their job information in their lookup tables from which they obtain all the main memory addresses used in the transfer of required data into their Local Store. Once done with their job, they signal the PPU. Depending on certain conditions, the activation function may be performed inline, or it may be performed synchronously as another step. If batch learning is enabled, weight updates are performed on the PPU at the end of a batch.

6.3 Detailed Implementation

In this section, the individual steps are given more detail. While originally intended to generalize to a common layer type, the fully-connected layer and convolution layer diverged enough in their implementation to necessitate separate sections. The simpler fully-connected layer is described first, followed by the convolution layer.

6.3.1 Conventions

The following acronyms are used in pseudo-code listings:

DMA (n.b.): Start of non-blocking DMA transfer.

LS_Rsv[(n)]: Reserve memory in the LS (Local Store) of size a function of *n;* not necessarily of size n. The technique for reserving memory is to increment running pointer into a globally declared array of a fixed size before copying the pointer value to the requested array identifier. As an example:

```
// Global
char fixed_size_array[MAX_SIZE];
char *ptr;

// Local. Allocate (reserve) space for n floats.
{
        float *a;
        a = ptr;
        ptr += n * sizeof(float);
}
```

Listing 13: Manual memory allocation on the SPEs

6.3.2 SPE Initialization (Common)

This step is fairly straightforward. For all available SPEs (the number is specified as a compiler directive), the SPE binary is uploaded, and execution begins. The SPEs enter the main processing loop waiting for a command. Because no inter-SPE communication is necessary, the problem state and LS areas are not memory mapped. The SPE context

creation, loading of the binary, and thread creation are all aided by the SPE library that is provided in the SDK.

6.3.3 Fully-Connected Layers

6.3.3.1 Layer Initialization and Representation

As described previously, from the implementation perspective, it is beneficial to represent the forward and Backpropagation procedures of the algorithm as matrix vector multiplications, in which the vector multiplicand is the output from the previous layer and the vector product is the output from the current layer representing every neuron on the layer. Similarly, the gradients and errors for a layer can be represented as arrays. This is the reason why most implementations of the MLP training algorithm follow an approach that caters to operations on layers, as opposed to on individual neurons. By doing so, data is organized into Structures of Arrays (SOA) as opposed to Arrays of Structures (AOS). The former method is recommended for the Cell Processor [53] and is used in this work.

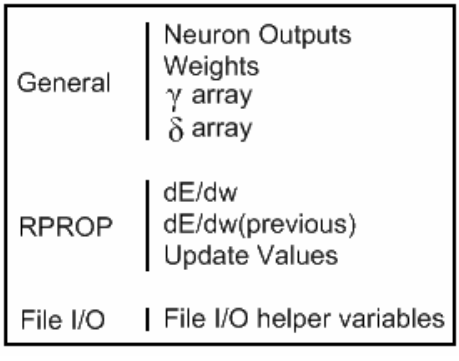
Initially, each fully-connected layer is created by supplying as parameters the number of neurons on that layer, the number of neurons on the layer just before it, and optional initialization settings. The only available option is the range of the randomized weights ([-0.005, 0.005] was used in the experiments). Alternatively, each layer may be read in from a file that was previously saved. Loading saved layers is useful since it can take many hours or even days to train a network. The data in the layer created includes a vector of outputs (neuron values) and a weight matrix (connections *into* each of the neurons on the layer). Recalling Chapter 2, Backpropagation involves the propagation of the error back through the network. For the remainder of this section, two new symbols will be used:

$$\gamma_n = \left(\frac{dE}{dx}\right)_n \qquad \delta_n = \left(\frac{dE}{dz}\right)_n \tag{6.5}$$

Backpropagation through fully-connected layers is a sequence of the following two operations:

$$a) \quad \bar{\gamma}_{n-1} \Leftarrow A^T \bar{\delta}_n \qquad b) \quad \bar{\delta}_{n-1} \Leftarrow F'_v\left(\bar{z}_{n-1}\right) \bullet \bar{\gamma}_{n-1} \tag{6.6}$$

As observed, two additional arrays need to be stored – namely the δ and γ vectors. Additional information stored is the attributes related to the RPROP adaptive training mode (if enabled), learning rate, and size of the total weight matrix (for file operations). The final product of this step is the *standard layer* as it is only good for the single-threaded version of the implementation.



s

Figure 6-1: Standard Fully-connected Layer

The standard layer needs to be reformatted and extended with additional information for the parallel multi-SPE implementation. The standard layer is converted into a SPE-*optimized layer* (or *SPE layer*) and encapsulated into a *SPE-optimized layer info* (or *SPE layer info*) object. The SPE layer becomes the global job to be partitioned into smaller jobs which are also placed within the SPE layer info object.

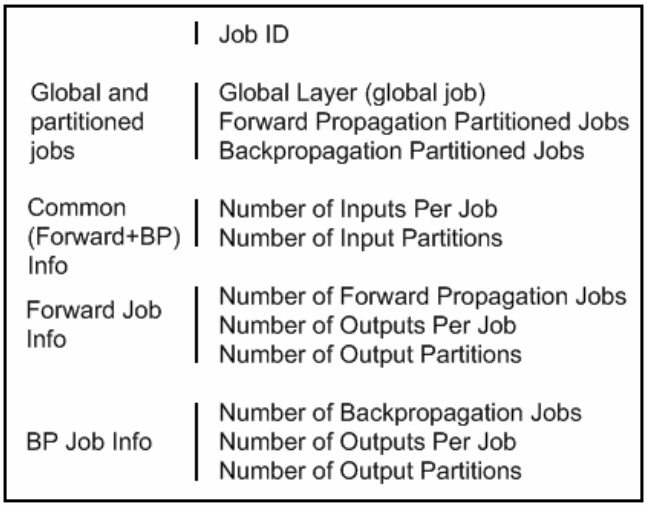
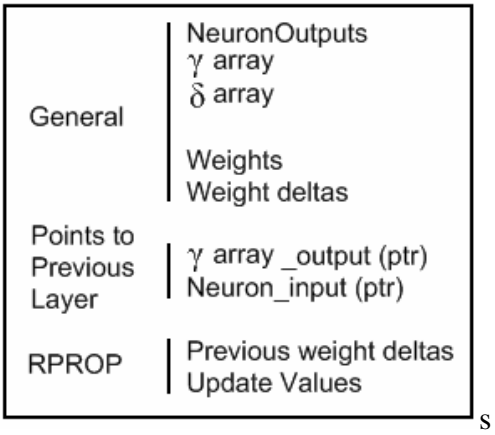


Figure 6-2: SPE Layer Info struct

Several steps are taken when generating the parameters for the SPE layer info. Initially, only the global job is generated by using the standard layer as an input parameter. Output, gradient, and error arrays are copied from the standard array into 128 byte aligned memory locations so that they meet the DMA criteria for improved transfer performance. Attributes, such as the number of neurons and learning rate are copied as well. The number of neurons is rounded up to the nearest multiple of four, and stored in a new attribute. The original value is not used by the SPEs but is kept for reference. By working with multiples of four, 4-way SIMD architecture on the SPEs can be utilized effectively. Fig. 6-3 shows a simplified SPE-optimized layer. Note that there are pointers in this structure that point to memory belonging to the previous layer. The process of setting these pointers is described in the next paragraph.



s

Figure 6-3: SPE Optimized Layer

Once the SPE Layer Info structures are created, they are connected to one another. Connecting two layers (left layer and right layer for this discussion) updates the attributes within the right layer info structure. Specifically, the global layer inside the info structure is given information about the number of inputs it has and the initial weight values. Additional memory is allocated that is used by the SPEs for various reasons. Connecting layers links up their pointers (Fig. 6-4). For example, the location of the neuron output values for the previous layer and the destination for the calculated weighted sums of deltas (for the previous layer) during Backpropagation are set.

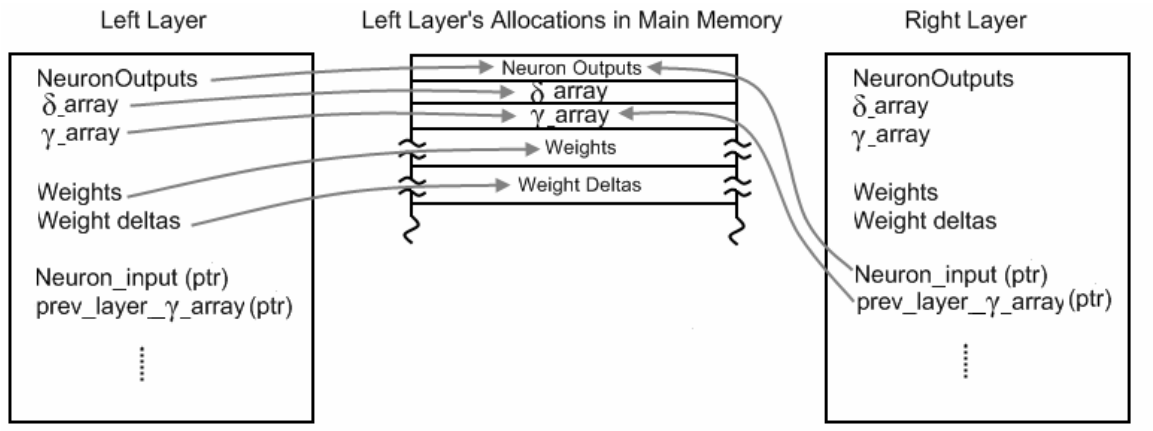


Figure 6-4: Connecting two layers

Having all the information within the global layer structure, it is now possible to partition it into smaller jobs. The process is done internally using an algorithm that takes into account the size of the weight matrix and number of available SPEs.

Parallelization is implemented on a per-layer basis. The partitioning scheme for fully-connected layers is done by block-wise partitioning of the weight matrix with each block being small enough to fit into the local store of an SPE. The number of jobs may exceed

the number of active SPEs, and thus a single SPE usually performs multiple jobs synchronously. The SPE layer info struct created contains high level information that is common to all jobs on that layer, including the number of jobs for forward propagation, number of jobs for back propagation, the number of rows and columns in each matrix block, and some allocated sandbox memory that the SPEs are given DMA access to.

This partitioning scheme has each SPE update a *partial* weighted sum of a part of the output vector. For example, in Fig. 6-5, given that an SPE performs the jobs containing Block B and Block C, it will have a partial weighted sum of the shaded portion of the outputs.

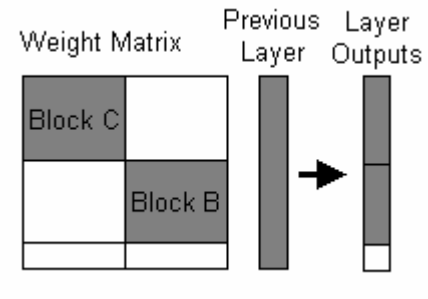


Figure 6-5: Weight Matrix partition such that inline activation cannot be performed

If on the other hand, a single SPE performs jobs containing Block A and Block B as in Fig. 6-6, it has the *full* weighted sum of the shaded region of the output neurons. Two advantages arise from the second case. Firstly, the partial output neuron values originating from the separate SPEs do not have to be summed together, which requires inter-SPE or SPE-PPE communication, since each SPE works on an exclusive region of memory and is guaranteed to have the final value for that portion once all jobs are completed. Secondly, the activation function can be performed inline on the individual output regions within the SPE code. As a counterexample, if the final outputs were only partial sums, one way of achieving the same result would be to have the PPU wait for all partial sums, then add them up, and then perform the activation function on the result as a separate step.

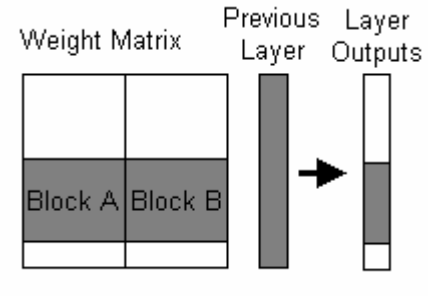


Figure 6-6: Weight matrix partition supporting inline activation

For this reason the job partition algorithm is biased toward distributing matrix blocks in row wise fashion. Other goals of the algorithm are to optimize local store memory usage by maximizing the size of each block, to keep the number of rows and columns a multiple of four, and to put as many SPEs into use as possible.

The partitioning heuristic algorithm devised is shown in Listing 14.

```
// Find the maximum number of inputs with the minimum number of outputs
// for each job.

Set the number of inputs per job to the maximum (number of neurons on the previous layer)
Set the number of outputs to the minimum so that the SIMD architecture could be utilized (4)
Set the input divisor to 1 (no divisions)

Start Loop
        Calculate memory requirement for Backpropagation on the LS given this
            combination of input and output sizes
        If the memory required does not exceed local store space
                Exit Loop and continue

        Increment the divisor
        Recalculate the number of inputs per job based on the new divisor
           - Make sure that the new number is a multiple of four (including the remainder)

        If the new number is the same as the previous
                Exit Loop and notify user that there is not enough local store space

// Choose the number of output divisions for forward propagation

// The following starting choice of the output divisor attempts to find a balance between
// using as many SPEs as possible and maximizing individual job size (reducing overhead)
Set the output divisor to (2*NUM_ACTIVE_SPES) / num_input_divisions
Calculate the number of outputs based on the output divisor
   - Make sure that the new number is a multiple of four (including the remainder)

Start Loop
        Calculate the memory requirement for Forward propagation given this
           combination of input and output sizes
        If the memory required does not exceed the local store space
                Exit Loop and continue

        Increment the divisor
        Recalculate the number of outputs per job based on the new divisor
           - Make sure that the new number is a multiple of four (including the remainder)

        If the new number is the same as the previous
                Exit Loop and notify user that there is not enough local store space

// Repeat the above loop for Backpropagation. The only difference in the calculation of
// the memory requirement. Backpropagation requires more space as a function of the
// number of inputs and outputs.
```

Listing 14: Job Generation for forward and Backpropagation for fully-connected layers

Once the job partitioning parameters are obtained, the weight matrix is reformatted, and the children jobs are created. Knowing the column width of each block, the matrix is reformatted so that block division can be used as shown in Fig. 6-7. Note that padding must be inserted so that the column width is divisible by four. No new memory is allocated for children jobs. Instead, each child job is given pointers into the parent job's memory portions.

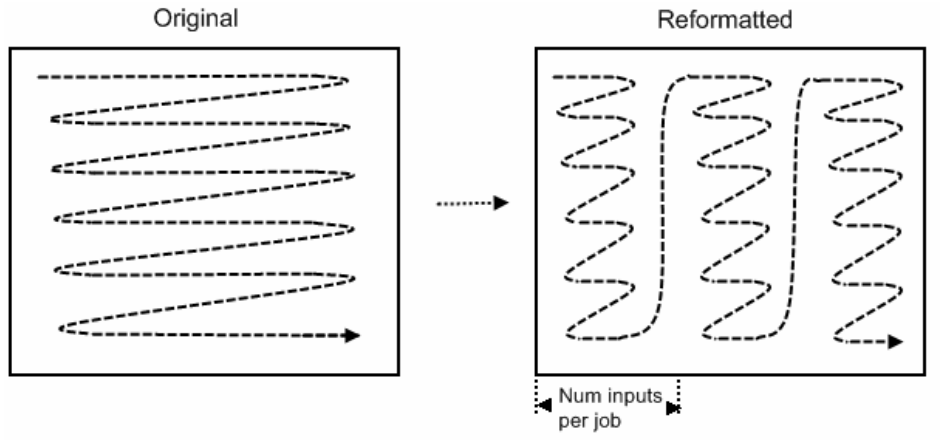
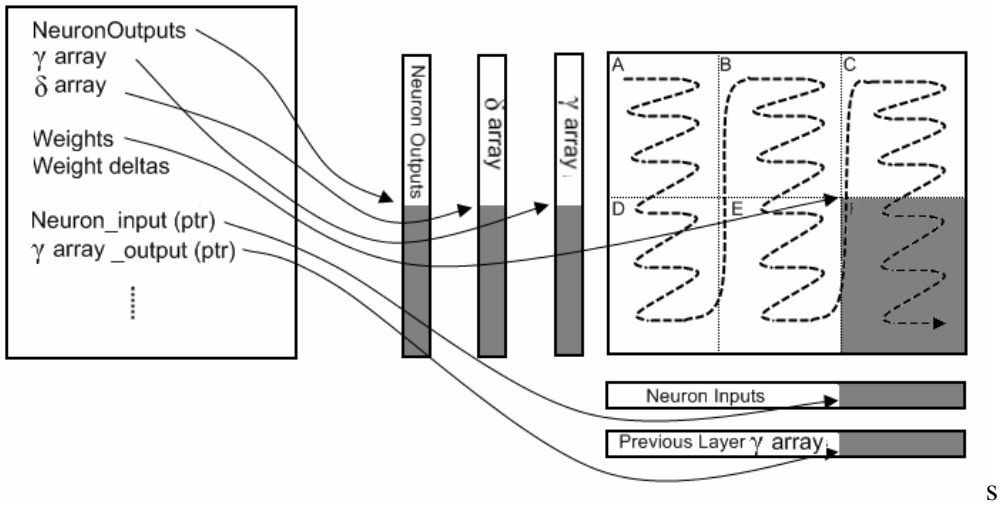


Figure 6-7: Reorganization of the weight matrix in main memory

Each child job's pointers are set to some location in the global job's allocated arrays. The location of these pointers is based on the input and output indices. An example is shown in Fig. 6-8.



s

Figure 6-8: One of several jobs for a fully-connected layer

6.3.3.2 SPE Initialization

Two convenient characteristics in the individual jobs that are created during initialization is that they do not change throughout the algorithm and that their completion time is deterministic. These are very useful characteristics as they do not require the use of dynamic job scheduling, which usually implies additional overhead. Instead, for each layer, an SPE is given a list of jobs that it is responsible. This list, along with several other attributes is stored in a local table, and is indexed by a global and unique layer id. As a consequence, when it comes to processing of a given layer, the SPEs simply receive the layer id and use it as a look up token within the table to obtain the information they need.

6.3.3.3 Forward Propagation

During forward propagation, when a fully-connected layer is to be processed, the PPE sends exactly one message, encapsulating the command and layer id, to each active SPE. Every message is the same and is shown in Fig. 6-9.

Figure 6-9: Fully-connected Forward Propagation Command

Once an SPE receives the command, it performs its portion of the work according to pseudo-code in Listing 15.

```
Obtain the EA and size of the job list from the local layer table
my_output_neurons ← LS_Rsv (partial or full* sum of output neurons)

While there are jobs remaining
        DMA up to MAX_JOBS_PER_SPU jobs from the job list into the job data buffer
        for each job in the job data buffer
                input_vector ← LS_Rsv(job specification)
                weight_matrix ← LS_Rsv(job specification)
                DMA the input vector and weight matrix into the local store
                Matrix Vector Multiplication (see Alg <>)
                If inline activation option is enabled*
                perform activation
                DMA (n.b.) the activated partial output vector into main memory
                Release memory for input and weight matrix
        Loop
Loop

If inline activation option is disabled
        DMA the output neuron partial (or full*) sums back into main memory
        Wait for DMA transfers to complete
Signal the PPE

*Applies if the global layer matrix was small enough to be separated row wise only. If so, each job results
in the completion of some part of the output vector. Otherwise, only partial sums are generated and need
to be summed on the PPE before performing activation.
```

Listing 15: Fully-connected Layer Forward Propagation SPE Pseudo-code

The corresponding PPE pseudo-code is shown in Listing 16.

```
For all SPEs: Send command (SPEs begin processing their jobs)
If inline activation is enabled, zero out global neuron array
For all SPEs
        Wait for completion of SPE
        If inline activation is disabled
                Add partial output neuron array values to global neuron array
Loop

If inline activation is disabled
        Perform activation on global neuron array as a separate step
```

Listing 16: Fully-connected Layer Forward Propagation PPE Pseudo-Code

The matrix-vector multiplication step uses the weight matrix and input array respectively. The implementation is optimized by unrolling of the outer loop and utilizing the SIMD instruction set as shown graphically in Fig. 6-10.

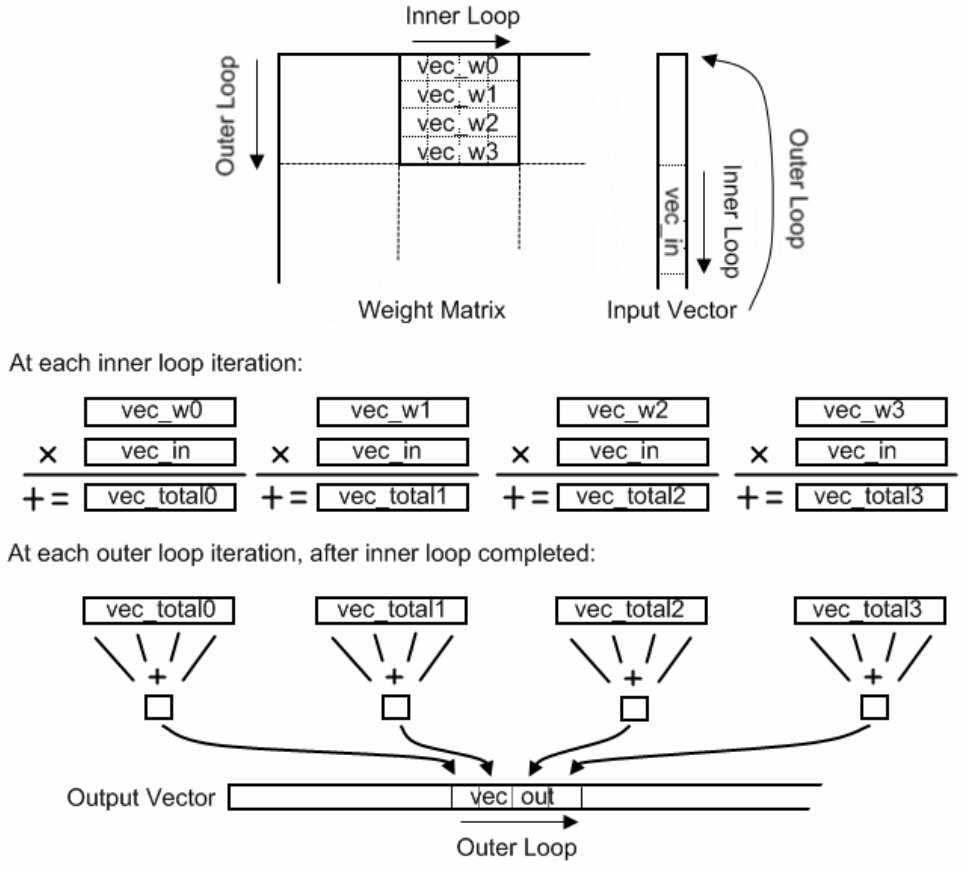


Figure 6-10: Matrix Vector multiplication

Each iteration of the outer loop performs four dot products at a time. The inner loop keeps a running sum of four partial sums of the four dot products. Once the inner loop

exits, each of the partial sum vectors is summed into a scalar value which is placed in the correct location of the output vector.

Once a job is processed and if inline activation is enabled, the activation function is used to process the portion of the output vector just calculated, and the result is transferred into main memory via DMA without blocking (each job has access to an exclusive portion of the output vector and thus no two jobs will modify the same portion of the output vector making it safe to transfer a portion once its corresponding job is done).

If inline activation is not enabled, then multiple jobs may access the same portion of the output vector. In this situation, a running partial sum is kept of the full output vector (size of the actual global layer). Each job is aware of the portion of this output vector that it needs to write to. Once all assigned jobs are processed, the full output vector is transferred into memory using a blocking DMA transfer. In this situation, the PPE receives the output vectors from each SPE, and sums them all up before executing the activation step separately.

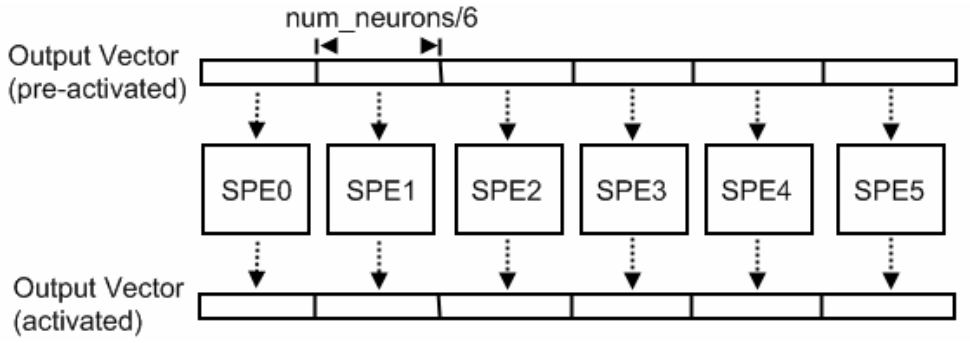


Figure 6-11: Optional distribution of activation function

The activation step simply transfers exclusive portions of the output vector to each SPE (Fig. 6-11). The SPEs use SIMD instructions to perform the activation on their vector portion and signal the SPE when they have transferred the results back into main memory.

At this point, the forward propagation through the fully-connected layer is complete.

6.3.3.4 Backpropagation

The SPE layer info struct also holds a list of jobs that are performed during Backpropagation. Because of the way that the matrix is represented in memory after being reformatted, the Backpropagation jobs need to have the same block width (number of inputs per job) as the forward propagation jobs. The block height (number of outputs per job), however, may be smaller due to the need for storing more data on the local store in the Backpropagation algorithm. It is this reason that a separate list of back propagation jobs is stored.

Similarly to the forward propagation step the message shown in Fig. 6-12 is sent to all SPEs.

FC Backprop Command (8 bits)	Layer ID (24 bits)

Figure 6-12: Fully-connected Backpropagation Command

The top-level pseudo code for the Backpropagation algorithm is shown in Listing 17. Recall that the γ for any neuron is obtained by getting the weighted sum of the δ's from the next layer. The δ is then calculated by multiplying the γ value by the derivative of the activation function with respect to the input. Also note that online learning is selected using a compiler directive and not during runtime. This way, there is no conditional branching required.

```
Obtain the EA and size of the job list from the local layer table
partial_delta_sum ← LS_Rsv (partial delta sums for the previous layer)

while there are jobs remaining
DMA up to MAX_JOBS_PER_SPU jobs from the job list into the job data buffer
        for each job in the job data buffer
                input_vector ← LS_Rsv(job.num_inputs)
                output_vectors ← LS_Rsv(job.num_outputs)
                weight_matrix ← LS_Rsv(job.num_inputs * job.num_outputs)
                weight_deltas ← LS_Rsv(job.num_inputs * job.num_outputs)
                delta_vector ← LS_Rsv(job.num_outputs)
                DMA in the input vector, output vector, and weight matrix
                If inline activation
                        DMA in the γ vector into the delta_vector
                        Obtain the δ vector by processing the γ using the derivative of
                           the activation function w.r.t. the inputs
                Else
                        DMA in the δ vector (already calculated in a previous step)
                If online learning, zero out the weight_delta array
                Else, DMA in the latest weight_delta from main memory
                Transpose Matrix Vector Multiplication (see Alg <>)
                   (modifies the weight_deltas matrix and wds for the previous layer)
                If online leaning,
                        update the weight matrix block for this job and send
                           back in the proper place in main memory
                Else
                        DMA the updated weight_deltas back to main memory
                Release all job related memory
        Loop
Loop

DMA the partial wds to main memory (results from each SPE are summed on the PPE)
Signal the PPE
```

Listing 17: Fully-connected layer Backpropagation SPE pseudo-code

The associated PPE pseudo-code follows:

```
If inline activation is disabled
        obtain the deltas by performing a separate step on the weighted delta sums

For all SPEs: Send command (SPEs begin processing their jobs)
Zero out the global wds on the previous layer
For all SPEs
        Wait for completion of SPE
Add partial wds result to global wds on previous layer
Loop
```

Listing 18: Fully-connected layer Backpropagation PPE pseudo-code

The first part of the algorithm depends on whether inline activation is enabled; this again depends on whether the global weight matrix for the layer was split vertically between jobs. If inline activation is enabled, the SPE performs a DMA transfer of the γ vector onto the LS and obtains the δ vector by performing an element by element product of the γ vector and the derivative of the activation function with respect to the input of the neuron at that iteration. Vector operations are used in this step. Otherwise, the δ's have already been computed by a separate step and are transferred onto the LS.

The second part of the Backpropagation algorithm is in some ways similar to the forward propagation algorithm in that it is a matrix vector multiplication. The matrix in this operation, however, is the transpose of the matrix block. The vector is now the delta array for the neurons on this layer associated with the current job. Instead of first performing the transpose and then performing the multiplication, the operations are reformulated to work directly with the weight matrix while still taking advantage of the SIMD units. As a separate advantage, it is not necessary to reserve additional space for the transpose of the matrix block. The concept is shown in Fig. 6-13. In the implementation, the outer loop was unrolled and code was optimized to reduce dependencies between steps.

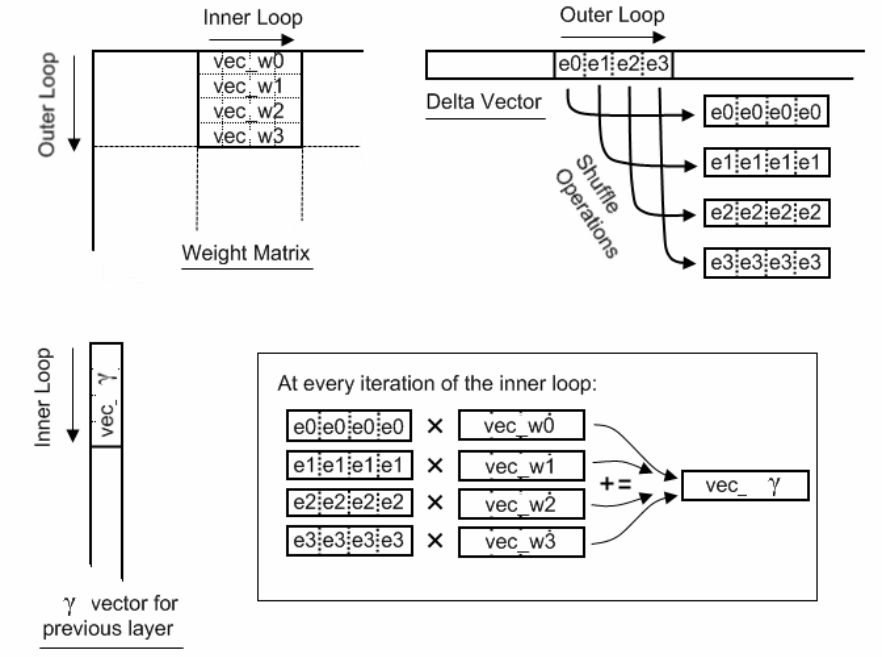
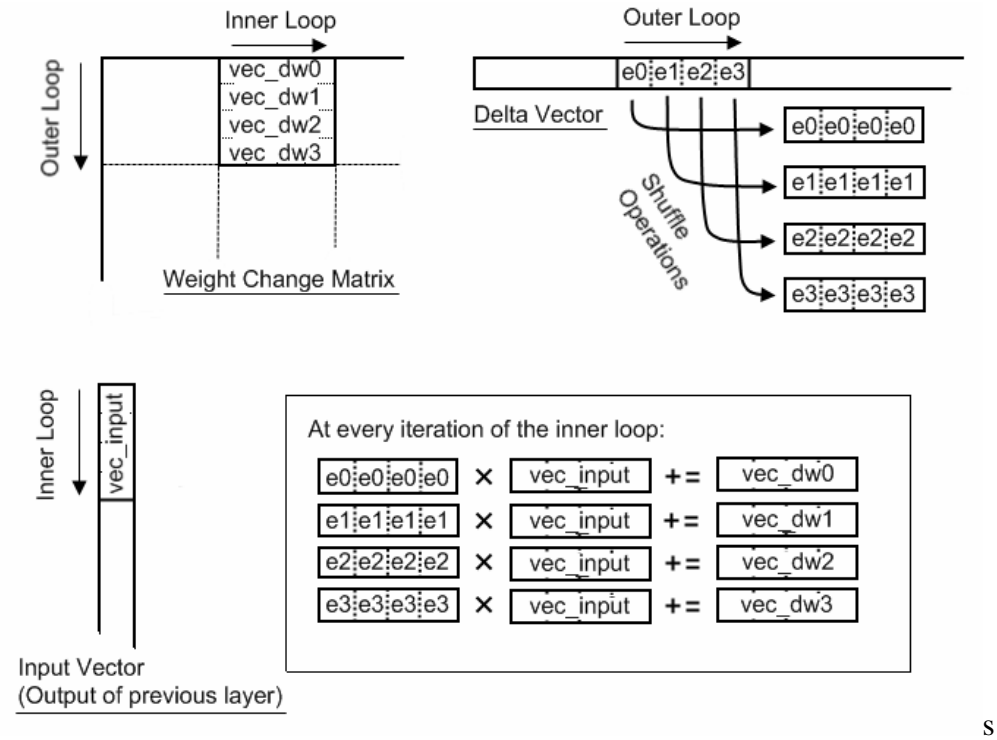


Figure 6-13: Matrix vector multiplication requiring no transpose to be taken

The updating of the changes in the weights was also performed in this inner loop. The procedure is illustrated in Fig. 6-14.



s

Figure 6-14: Updating of weights

If incremental learning is enabled, the next step is to update the weights in the matrix block and send the result back into main memory. The matrix block update is easily performed using SIMD multiply-and-subtract operations using the weight deltas and learning rate vector that is obtained by copying the learning rate into each of the four vector elements. If incremental learning is not enabled, the updated weight deltas are transferred back to main memory.

Once all jobs are completed, the γ vector for the previous layer is transferred into main memory for consumption by the previous layer.

6.3.4 Convolution Layers

The procedures for initializing the general convolution layer and converting it into an SPE-optimized info structure, as well as the forward and Backpropagation processes are similar to those of the fully-connected layer. However, the actual data present in each object and the methodology for dividing the workload differ significantly as will be detailed in this section. To reduce redundancy, some portions of this section may refer to the fully-connected section.

In the implementation, both layer types may be used in a single network. Going forward, convolution layers may be connected to fully-connected layers but not vice versa. Also, for simplicity of design, for any two connected convolution layers, every kernel operates on every input feature map. LeCun experimented by performing custom connections to see which worked best [12]. "Fully-connected" convolution layers performed very well as well.

6.3.4.1 Layer Initialization and Representation

Several analogies can be made between the fully-connected layer and convolution layer. The neurons, and outputs, on a convolution layer are termed *feature maps* and represent the filtered output images after application of a kernel matrix to each of the input feature maps (from the previous layer). The number of feature maps is specified as a network parameter and equals the number of *kernel weight matrices* (one kernel per output feature map). Kernel weight matrices, or kernels, are analogous to the weight matrix.

Creating a *general convolution layer* involves specifying the kernel length, the number of output feature maps on the layer, the height and width of each of the input feature maps from the previous layer, and specifying if random weights in the range [-0.005, 0.005] are to be used. Alternatively, layers may be loaded from a file to restore previously learned kernel weights. The general convolution layer, therefore, has a contiguous array for output feature maps, a contiguous array for kernels, an array for the γ and δ vectors used in Backpropagation, and attributes for the kernel length, number of output feature maps, and height and width of each output feature map (calculated automatically based on height and width of input feature maps and kernel size). Additional arrays and attributes for RPROP Backpropagation are also included. Fig. 6-15 displays a simplified general convolution layer.

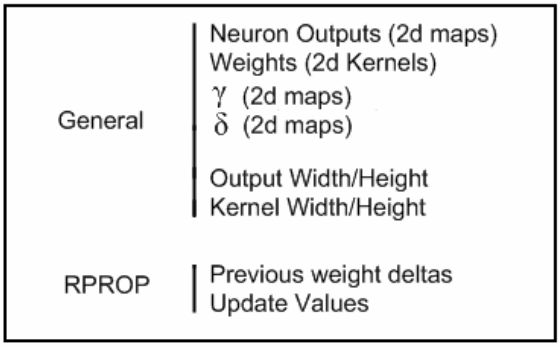


Figure 6-15: General Convolution Layer

An SPE-optimized layer information structure is created from a general convolution layer in the same manner as before. The first step generates the global SPE layer job (Fig. 6-16) within the SPE layer information structure by copying array elements and attributes from the general layer structure.

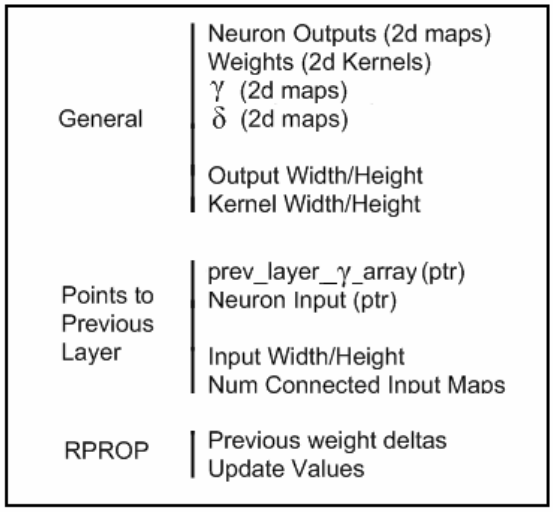


Figure 6-16: Optimized Convolution Layer

Because all the arrays represent two-dimensional structures (i.e.: kernel matrix, feature map), each row of every structure is padded to a multiple of four due to implementation details stemming from DMA requirements and Vector operations. An example of a padded 2-d map is shown in Fig. 6-17.

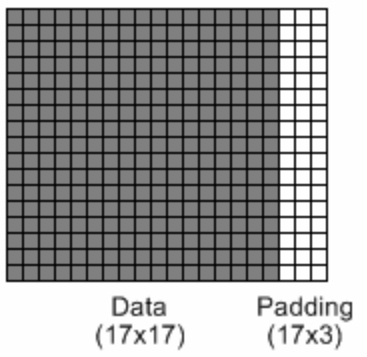


Figure 6-17: Padded Feature Map

Another step, reasoning for which will become apparent shortly, is to generate four copies of the kernel matrix, each shifted by a different amount as shown in Fig. 6-18.

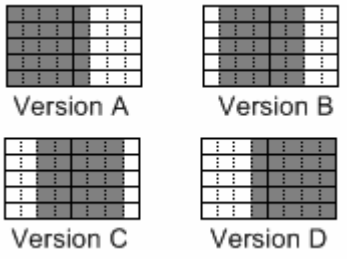


Figure 6-18: Example of a 5x5 convolution kernel as it is stored in memory. Four identical vecsions are generated, each with a different column offset.

Next, adjacent layers in the network are connected, which fills out the remaining attributes of the global job by linking up the inputs and outputs between the layers as shown in Fig. 6-19. Finally, the global job is partitioned into smaller jobs.

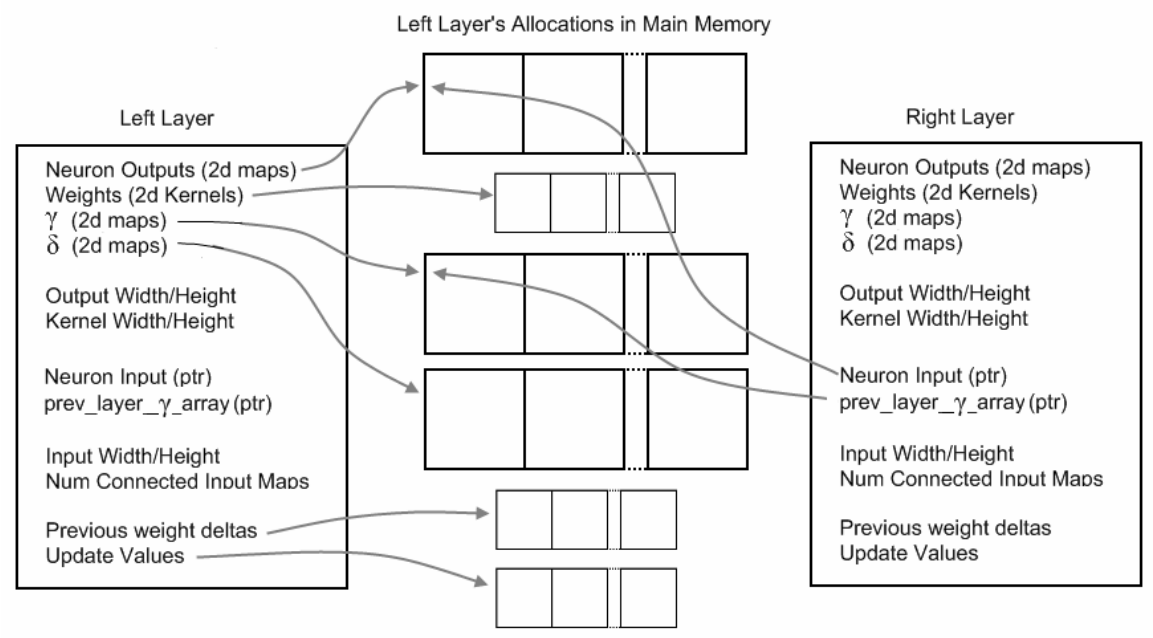


Figure 6-19: Connecting two convolution layers

The method for partitioning of the global job differs from the method used in the case of the fully-connected layer. A major factor in the partitioning scheme for the fully-connected layers was that each job fits into the limited memory of the local store of an SPE. This requirement is markedly relaxed when dealing with convolution layers since the weight kernels take up significantly less space. For example, a typical kernel has a size of 5x5 (8x5 when padded). Recalling that there are four versions of each kernel, having 50 feature maps implies 200 kernels for a total weight count of 5000. The space required would be 20 Kb if using 32 bit floating point numbers. With storage space not being a limiting factor in most cases, the number of jobs created is always less than or equal to the number of available SPEs.

Recalling convolution layer architecture, every output feature map is influenced by every input feature map through a corresponding kernel. It is logical, therefore, to partition the global job by assigning a subset of the output and/or input feature maps to each child job. The task is to decide on a strategy. Now, recalling the goals for partitioning fully-connected layers, inline activation is possible only if an SPE has exclusive access to a portion of the output array – that is: it computes the final values locally. It makes sense therefore to partition convolution jobs such that output feature map exclusivity is attained. This is done by assigning all input feature maps to each SPE, but only a subset of the output feature maps. As expected, iterating over all input feature maps produces the final pre-activation values of the owned output feature maps and the activation function can be performed locally on the local store. In conclusion, the partitioning algorithm simply divides the total number of output feature maps by the total number of available SPEs and assigns the resulting number to each SPE (accounting for the remainder). All SPE jobs include all input feature maps. An assumption is made that a job will not exceed the space of the local store. Fig. 6-20 summarizes the concept.

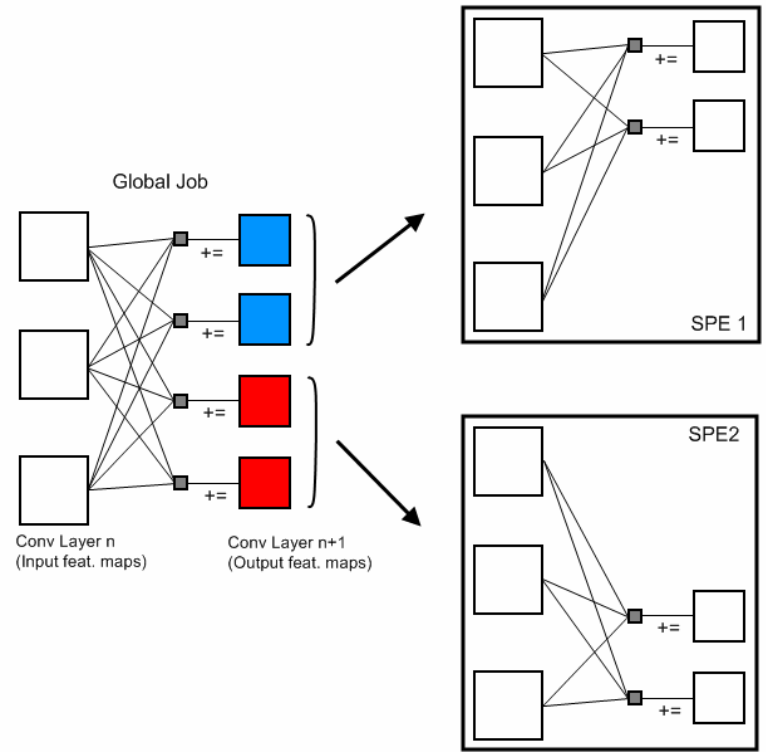


Figure 6-20: Job partitioning for convolution layers

SPE and layer unique job information is uploaded and stored in the local store of each SPE during initialization in the same manner as for the fully-connected layers. Due to different sizes of fully-connected jobs and convolution jobs, two separate tables are used. Layer IDs might, therefore, overlap between the two versions.

6.3.4.2 Forward Propagation

Again, processing a convolution layer is initiated by sending a single command to each SPE (Fig. 6-21).

CV Forward Command (8 bits)	Layer ID (24 bits)

Figure 6-21: Convolution Forward Command

The job handling overhead is much smaller due to there being exactly one job per SPE. The pseudo-code for the SPE, once the command is received is as follows:

```
input_feature_maps ← LS_Rsv(num_input_maps * size_one_input_fm)
kernel_weights ← LS_Rsv(num_output_maps * 4 * size_one_kernel)
output_feature_maps ← LS_Rsv( num_output_maps * size_one_output_fm)

DMA in input_feature_maps and kernel_weights
output_feature_maps ← 0…

For every output feature map: fm_out
        For every input feature map: fm_in
                Update fm_out (See Algorithm <>)
        Loop
Loop

If inline activation is enabled
        Perform activation on entire output_feature_maps array
DMA out output_feature_maps
Signal the PPE
```

Listing 19: Convolution forward propagation SPE pseudo-code

The PPE pseudo-code is exactly the same as the fully-connected pseudo code for the forward propagation case:

```
For all SPEs: Send command (SPEs begin processing their jobs)
If inline activation is enabled, zero out global neuron array
For all SPEs
        Wait for completion of SPE
        If inline activation is disabled
                Add partial output neuron array values to global neuron array
Repeat

If inline activation is disabled
        Perform activation on global neuron array as a separate step
```

Listing 20: Convolution forward propagation PPE pseudo-code

Unlike in the fully-connected layer, updating the output neurons (output feature maps) is not performed using a matrix vector multiplication. While technically possible, the repetition of weight values would make it very inefficient in terms of storage.

Figure 6-22: Convolution operation

Updating an output feature map is performed by traversing an *n*x*n* kernel across the current input feature map from left to right, top to bottom, one pixel at a time as shown in Fig. 6-22. This is known as the convolution operation in image processing. On scalar

CPU architectures, this is easily done using systematic array indexing. However, to utilize the SIMD hardware of the SPE, a new method needed to be devised.

As mentioned before, four versions of the weight matrix are generated for every one matrix in the general layer structure such that each one is offset by different amounts. Because there are four elements per vector, the kernel's rows are padded to a multiple of four with 0's.

For example, as shown in Fig. 6-23, if the kernel size is 5x5, two vectors are used per row for a total of 8 values per row. There are four versions of the kernel. In the first version in each row the first vector takes on the first four values, and the second vector has the fifth kernel value followed by three 0's. In the second version, the first value of the first vector in each row is 0 followed by 3 values. The second vector of each row now has two valid values, followed by two 0's. The third version's first vector in each row has two 0's followed by two valid values. The second vector of each row in the third version has three valid values followed by one 0's. The fourth version follows the same pattern, with the first vectors starting with three 0's followed by one valid value and the second vectors having all valid values.

Having these four versions allows the algorithm to iterate over vectors (four elements) from left-to-right, as seen in Fig. 6-23, instead of one pixel at a time. At each vector location all four versions of the kernel are applied. This process allows for bigger loop blocks and utilizes the vector operations provided by the SPEs.

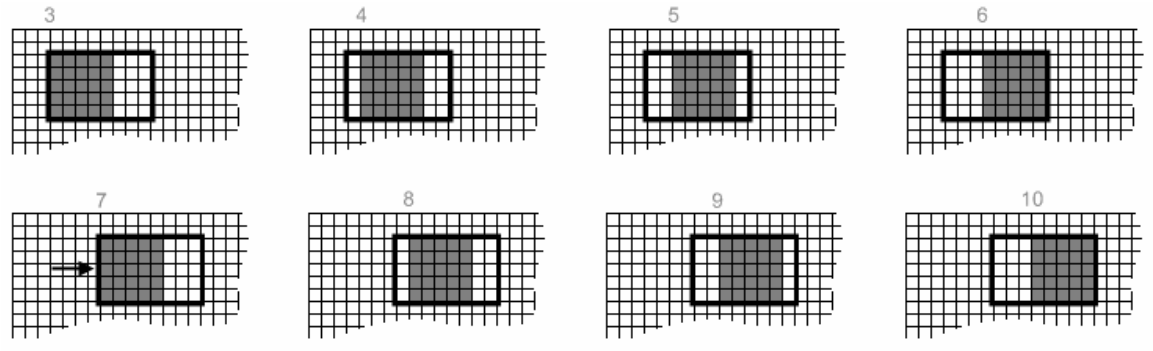


Figure 6-23: SIMD-optimized convolution

An important option that is more often enabled than it is not is subsampling. Subsampling involves taking steps of two pixels at a time, as opposed to one when traversing the kernel over the input feature map. The implementation in this work supports subsampling by only utilizing two of the four copies of the kernel matrix – version A and version C. Because several changes in indexing needed to be made, the option of subsampling can only be enabled or disabled by changing a compiler directive and recompiling.

Fig. 6-24 shows a graphical representation of the processing of a single patch of the input feature map. The patch travels across the input feature map from left to right, top to bottom. All four kernel versions (A, B, C, and D) are applied to the current patch at the

same time. The actual implementation in the software is unrolled and optimized for dependency reduction.

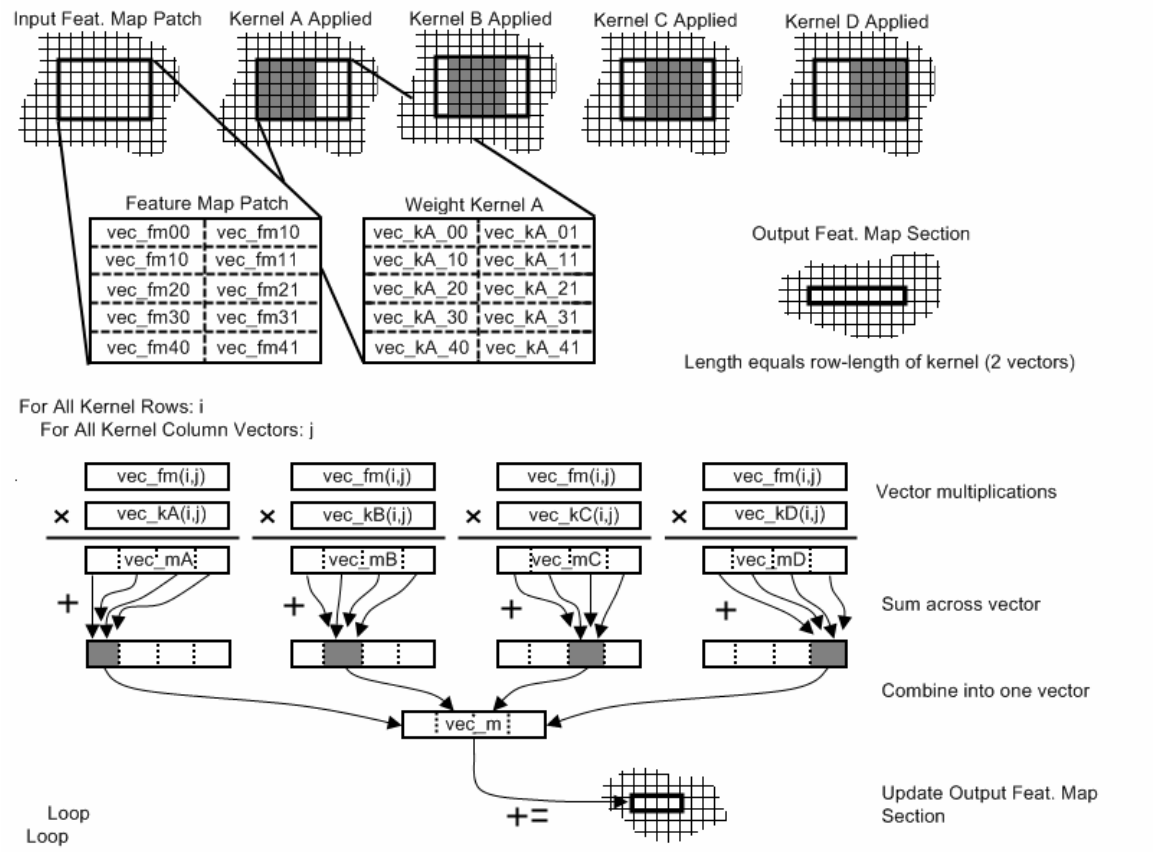


Figure 6-24: Detailed SIMD-optimized convolution

The figure represents forward propagation for the case that subsampling is disabled. When enabled, only two of the kernels (A and C) are used, and the output feature map is traversed at half the speed (since it is about half the size). The implementation is more complex as it involves careful shuffling of results into the proper location of the output feature map.

6.3.4.3 Backpropagation

The command sent to each layer follows the standard pattern:

CV Backprop Command (8 bits)	Layer ID (24 bits)

Figure 6-25: Convolution Backpropagation Command

The pseudo-code for the SPE follows:

```
input_feature_maps          ← LS_Rsv(num_input_maps * size_one_input_fm)
kernel_weights     ← LS_Rsv(num_output_maps * 4 * size_one_kernel)
dE_wrt_w_maps  ← LS_Rsv(num_output_maps * 4 * size_one_kernel)
dE_wrt_w_total   ← LS_Rsv(num_output_maps * size_one_kernel)
γ_map              ← LS_Rsv(num_output_maps * size_one_output_map)
δ_map_prev         ← LS_Rsv(num_input_maps * size_one_input_map)
If inline activation enabled,
output_feature_maps ← LS_Rsv( num_output_maps * size_one_output_fm)
DMA in input_feature_maps, kernel_weights, kernel_masks*
If inline activation enabled,
        DMA in δ array of this layer into γ_map
        DMA in output_feature_maps
Else
        DMA in γ array of this layer into γ_map
If using RPROP,
        DMA in the dE_wrt_w_total
Else
        dE_wrt_w_total ← 0
dE_wrt_x_prev, dE_wrt_w ← 0…
If inline activation enabled,
        generatel γ_map from current δ values in γ_map (derivative of activation function)
For every output feature map: fm_out
        For every input feature map: fm_in
                Update δ_map_prev
                Update dE_wrt_w_maps
        Loop
        dE_wrt_w_total += version A, B, C, and D of dE_wrt_w_maps
        If incremental learning,
                Multiply all elements of dE_wrt_w_total by learning rate
Loop

DMA out dE_wrt_w_total and dE_wrt_x_prev
Signal the PPE
```

Listing 21: Convolution Backpropagation SPE pseudo-code

The PPE pseudo-code is shown below:

```
If inline activation is disabled,
perform SPE-optimized derivative activation function
For all SPEs: Send command (SPEs begin processing their jobs)
Zero out δ_map_prev for the global job
For all SPEs
        Wait for completion of SPE
        If this is not the first layer,
                Add partial δ_map_prev from job to δ_map_prev of global job
        If online learning is enabled,
                Use the returned dE_wrt_w_total values to update all the weights
Repeat
If online learning is enabled,
        Generate updated four kernel versions (A, B, C, D) for each kernel in the layer
```

Listing 22: Convolution Backpropagation PPE pseudo-code

The first step is to obtain δ from the γ values that were calculated by the next adjacent layer (closer to the output). This is easily handled by performing SIMD operations on the entire δ map, which is represented as one long array.

During forward propagation, every pixel on an output feature map is generated by summing up the convolution results of the same coordinates of all the input feature maps. This is demonstrated graphically in Fig. 6-26.

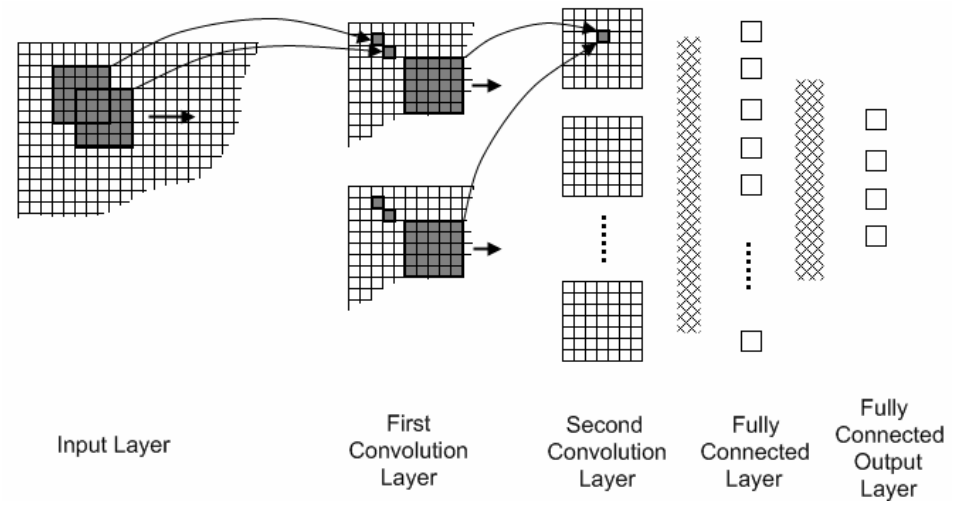


Figure 6-26: Convolution Layer forward propagation

A pixel on an output feature map is analogous to a single neuron in a fully-connected layer, which implies that each pixel has a δ value associated with it. The task, therefore, becomes to backpropagate every pixel's δ value to the corresponding pixels (neurons) of each of the input feature maps which had an effect on the output pixel during forward propagation.

The implementation traverses over the dE/dx map one vector (four elements at a time). At each selection, the four vector elements are each expanded into separate matrices – A, B, C, and D. In the actual implementation, only one row of the matrix is generated since each row is exactly the same. For the purposes of explanation matrices will be used. Fig. 6-27 shows the process.

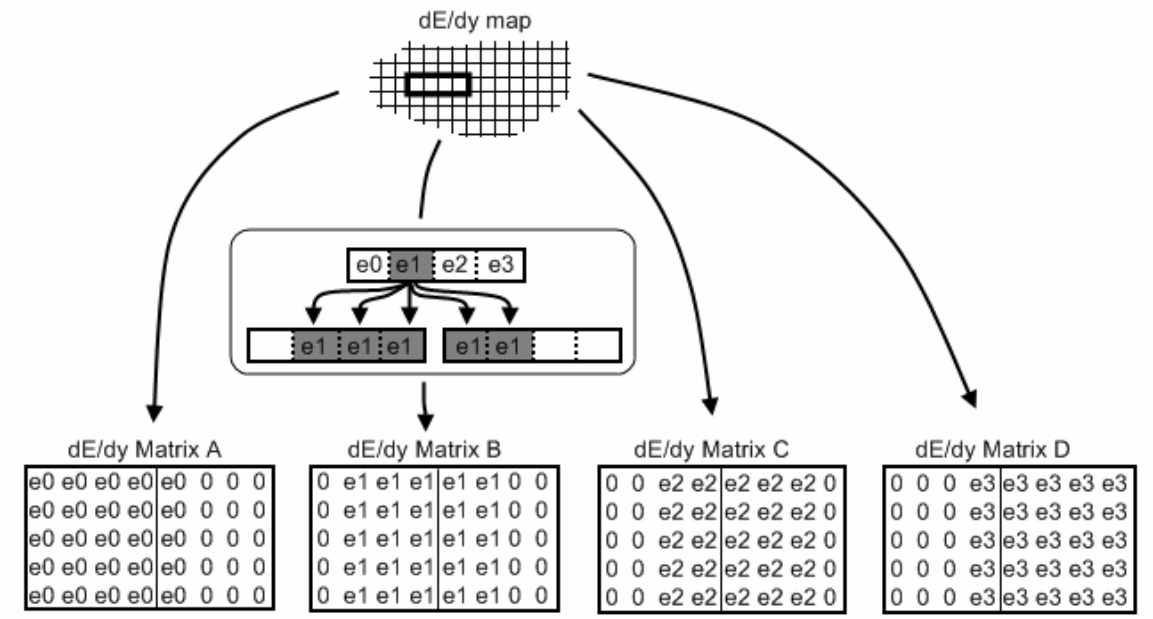


Figure 6-27: Generating the γ matrices

Next, the δ values are backpropagated "through" kernels by superimposing each version of the γ matrix to the corresponding kernel matrix. The two matrices are multiplied element by element (SIMD operations) and the result is added to the proper patch of each of the γ maps for the previous layer. Fig. 2-28 shows the process for one input and output feature map pair.

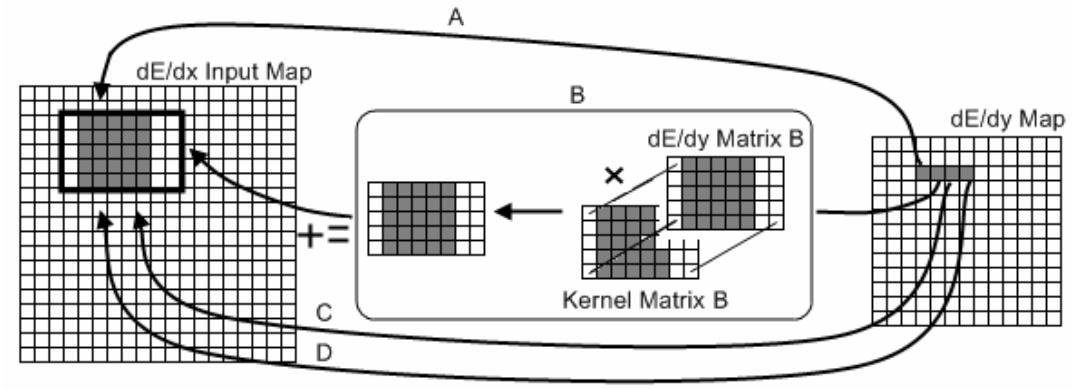


Figure 6-28: Convolution Layer Backpropagation

A separate array is used to keep track of the *dE/dw* values. Just as the kernel matrices, there are four versions of the *dE/dw* matrix. If RPROP is used, the values placed into the array are loaded from main memory. Otherwise, the contents of the array are zeroed. Each version of each kernel has an associated *dE/dw* matrix. Recalling the Backpropagation algorithm as described in Chapter 2, *dE/dw* is obtained by multiplying δ of the current neuron by the output of the neuron on the other side of the weighted connection (previous layer).

It is convenient that indexing into the input feature maps corresponds with the indexing into the γ maps of the previous layer because the updating of the *dE/dw* matrices can be performed in the same innermost loop. The process is shown in Fig. 6-29.

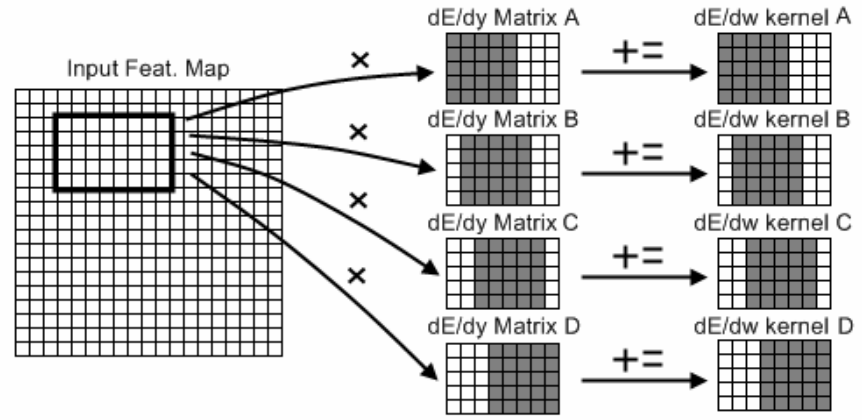


Figure 6-29: Convolution layer kernel updating

Before starting processing of the next output feature map, the four γ matrices are aligned and summed into one matrix. If incremental learning is enabled, the aligned *dE/dw* matrix is multiplied by the learning rate. The actual updating of the weight matrices is performed on the PPE once the SPE is finished. Otherwise, the matrix is left as it is.

Again, if subsampling is used, the indexing scheme is modified. While the same methods are used as stated above, the intricacies of the algorithm become somewhat more complex.

Once all output feature maps are processed, the SPE transfers the *dE/dw* matrix and γ map (for the previous layer in the network) to main memory. The PPE sums up the γ maps from each SPE by utilizing the Altivec engine. If incremental learning is enabled, the weights are updated using the obtained *dE/dw* values, and the new weight kernels are regenerated for use by the SPEs by recreating the four kernel versions (A, B, C, and D) for each one.

As always, the loops in this algorithm were heavily unrolled, and the usage of variables was optimized to minimize data dependencies and stalls.

Chapter 7: Implementation of Support Vector Machines on the Cell Processor

7.1 Chapter Introduction

As established in Chapter 3, there are multiple techniques for training Support Vector Machines (SVMs) including the popular working set technique, the sequential minimal optimization technique (an extreme case of the working set technique), and the cascade SVM and its variations. Each technique and its implementations have been developed to target and correct some unfavorable characteristic. The working set technique relaxes memory requirements, the SMO technique further reduces memory requirements and drastically simplifies the algorithm, and the Cascade SVM introduces parallelism via independent problem solving. Beyond these methods, other techniques exist that claim to improve accuracy and decrease training time.

This work did not try to develop a new algorithm altogether. Instead, the purpose was to examine existing implementations in literature and pick the ones that were considered a good fit for the architecture of the Cell Processor and the style of its programming models. Two implementations – the Gradient Projection-based Decomposition Technique (GPDT) and Cascade SVM – were selected, mostly for their focus on parallelization. Parallel hardware has only recently become widespread as many consider that the current processing technology is beginning to reach its limits in terms of transistor size and clock speeds. Unlike the MLP, SVM training does not easily break down into matrix multiplications, or any other inherently parallelizable process. The GPDT and Cascade SVM are among the first to add parallelization potential and were deemed good candidates for the Cell Processor.

This chapter explains the implementation details of the GPDT and Cascade SVM on the Cell Processor. Similarly to the previous chapter, many of the concepts are described using images rather than pseudo-code whenever possible. The order in which the chapter is laid out may not follow closely with the order of development as it was written with comprehension in mind.

7.2 Parallel Gradient Projection-based Decomposition Technique

The implementation of the GPDT is based on the freely available Parallel GPDT (PGPDT) source code that is licensed under General Public License (GPL). The GPL license allows for the modification of the code for experimental and educational

purposes. In this work, heavy modifications were placed into those parts which were most computationally intensive, while those parts which had little impact on the overall running time were kept so as to reduce sources of bugs. The final product consists mostly of original custom code and uses heavily optimized data representations.

As a simplified high-level review, the PGPDT algorithm is structured as shown in Fig. 7-1. The shaded portion of the figure is that which has been optimized in this work.

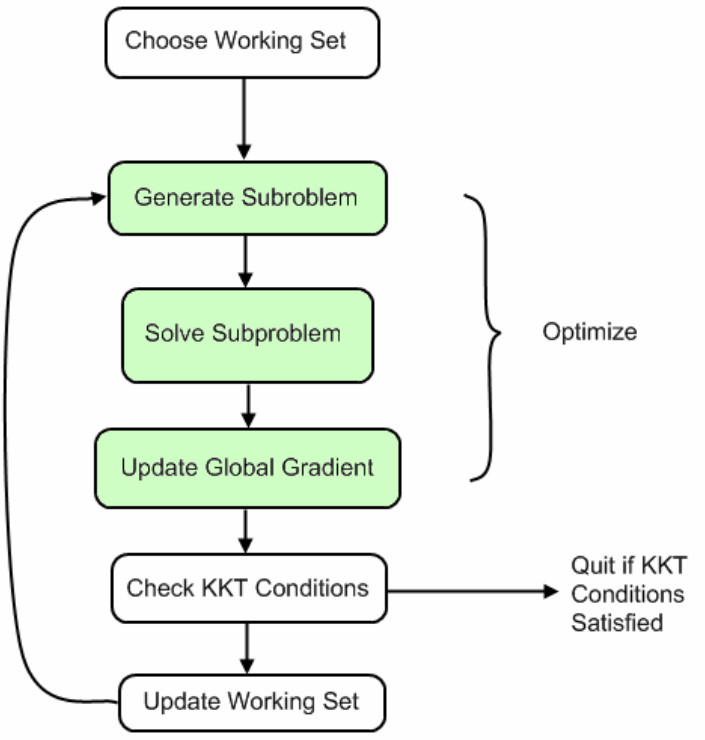


Figure 7-1: Parallel Gradient Projection-based Decomposition Technique

Of the major steps in the PGPDT loop, updating the gradient of the objective function in terms of the Lagrange Multipliers usually takes the most amount of time, followed by generating and solving the subproblem based on the current training vector working set. The selection of the subsequent working set has not been modified as it is insignificant in terms of the total running time. The caching access points have been modified, but the cache handling logic has remained the same.

The sections following begin to dig deeper into the low level details of the individual steps of the Cell implementation. Where applicable, differences between the Cell and original implementations are brought to light.

7.2.1 Training Data Representation

In the original PGPDT source code, all training input vectors are represented in memory as sparse vectors. The sKernel class kept track of these vectors using several data members:

```
int    lx[ell];          // the number of non-zero elements in each vector
int    *ix[ell];         // the indices of the non-zero elements
float  *x[ell];          // the actual values of the non-zero elements
```

The memory layout of this data is shown graphically in Fig. 7-2.

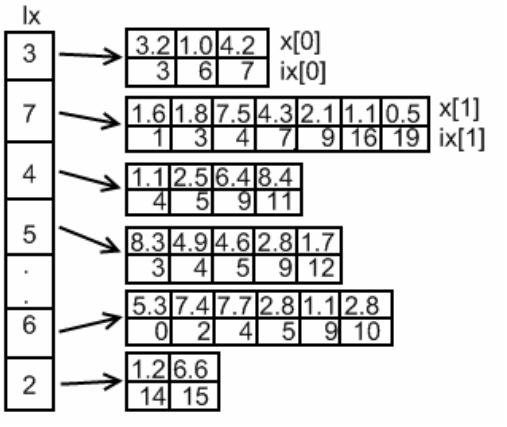


Figure 7-2: Original Training Set (Sparse Vector) data representation

To efficiently utilize the SIMD architecture of the SPEs, the data was converted from sparse arrays into 4-element sparse blocks (which is the vector size for the float datatype on the SPEs). Any 4-element *vec_float4* type which has at least one non-zero element is stored in memory. A non-zero block is therefore defined as a 4-element block in which at least one element is non-zero. Sparse index numbers represent an entire block instead of a single value. There is some padding necessary as well for optimizing DMA transfers. The new data representation has a much better fit for the SPEs at the expense of a slightly greater memory requirement. The new data members are:

```
int    vlx[ell];         // the number of non-zero vec_float4 elements in each training input
int    *vixm;            // the indices of non-zero vec_float4 elements (contiguous in memory)
int    *vix[ell];        // pointers into vixm for each input vector
vec_float4  *vec_xm; // the non-zero 4-element blocks (contiguous in memory)
vec_float4  **vec_x;  // pointers into vec_xm for each input vector
```

For example, the sparse array in Fig. 7-2 can be represented using the new format as shown in Fig. 7-3.

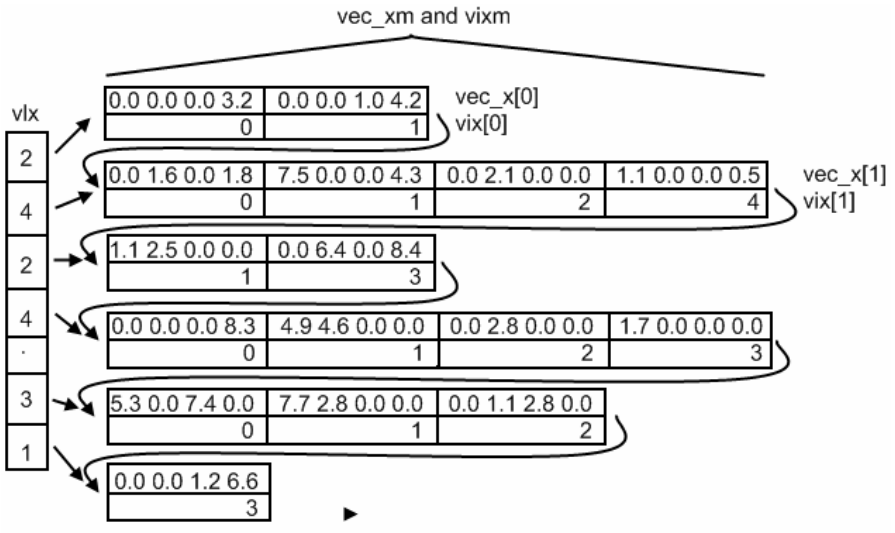


Figure 7-3: SIMD-optimized training set data representation

All references to sparse input vectors in the sections below refer to this sparse-block representation.

7.2.2 Generating and Solving the Subproblem

The first item that was focused on in this thesis was the subproblem QP solver. The original PGPDT source code was written in C and C++ and implemented the QP solver as a separate module, making it relatively easy to replace with a modified one. For this reason, this portion of the algorithm was targeted first. The subproblem solver attempts to find a solution to the problem 7.1.

$$\begin{aligned} &\min_{w\in\Omega} f(w) = \frac{1}{2} w^T A w + b^T w \\ &A \in \Re^{n_{sp} * n_{sp}},\ w, b \in \Re^{n_{sp}} \\ &\Omega = \left\{ w \in \Re^{n_{sp}},\ 0 \le w \le u,\ y^T w = e \right\},\quad u, c \in \Re^{n_{sp}} \end{aligned} \tag{7.1}$$

Recall from Chapter 3 that the full matrix of the entire problem as well as the other data are subdivided into subproblem data as follows:

$$G = \begin{bmatrix} G_{\beta\beta} & G_{\beta\delta} \\ G_{\delta\beta} & G_{\delta\delta} \end{bmatrix}, \qquad \alpha = \begin{bmatrix} \alpha_\beta \\ \alpha_\delta \end{bmatrix}, \qquad y = \begin{bmatrix} y_\beta \\ y_\delta \end{bmatrix}. \tag{7.2}$$

In problem 7.1, A is $G_{\beta\beta}$, w is α_β, and y is y_β.

7.2.2.1 Revision 1

The first revision of the solver module was written to take advantage of the SPE SIMD instruction set, reduce branches, unroll and clump loops together, and conform to the

general SPE programming guidelines. The inputs and outputs of this first version of the solver module were:

Inputs		**Outputs**	
n (<200)	*size of problem*	x[n]	*final alpha values*
A[n*n]	*kernel matrix*	ls[1]	*number of line searches*
b[n]	*linear portion*	proj[1]	*number of projections*
y[n]	*output class*	iter[1]	*total number of iterations*
x[n]	*initial alpha values*		
u	*u parameter*		
e	*e parameter*		
tol[1]	*tolerance*		

Table 4: Inputs and outputs of first revision of SPE solver

1. These parameters are specific to the GPDT algorithm.

The simplest way to plug an SPE module into existing code is to write an interface function that runs on the PPE and replaces the original single threaded QP function. This is called the function offload programming model. The interface function simply farms out the work onto the SPEs, usually by data parallelization, and waits until they are completed. The function offload technique was implemented in this work. The basic pseudo code for the interface function written is shown in Listing 23.

```
Collect all input data into a SPE-optimized structure.
- Align all arrays into 128 byte memory, pad if necessary.

If the SPE is not running, start it
Send the memory address of the SPE struct to the SPE

Wait for SPE completion
Move result from SPE struct back into original x[n] input
Clean up

Return
```

Listing 23: PPE Interface function for Revision I of subproblem solver

The SPE module, once started, communicates directly with the interface function by waiting and processing of commands via the Cell mailbox functionality. An overview of the SPE steps is shown in Listing 24.

```
Wait for address of data structure
DMA in the structure
Manually organize memory that will be needed
DMA all arrays into the LS

Execute the algorithm

Place results back into main memory
Signal the PPE
```

Listing 24: SPE module for Revision I of subproblem solver

While the SPE module was retained, and found to be useful later in the Cascade SVM implementation, as will be described in the proper sections, it was quickly identified that there were several limitations of this initial revision. First, there was quite a bit of overhead associated with the copying and restructuring of data at every invocation of the interface function. Second, due to the SPEs' limited room on the local store, only a kernel matrix of size ~200 by 200 could fit – resulting in a maximal subproblem size of 200. The GPDT algorithm was underutilized as it was designed to work on subproblem sizes of O(10^2) and O(10^3). Third, it would be quite difficult to parallelize this module across multiple SPEs since only one solver is allowed to run at a time in the working set technique. Fourth, the SPE module did not perform any subproblem generation, leaving a lot of the heavy work to the PPE. And lastly, any other portions of the algorithm would either take up more space on the SPE, reducing the total subproblem size, or require the reservation of their own SPEs, keeping the number of active SPEs below maximum.

During the course of actual development, work was temporarily halted on the QP solver while focus was placed on the gradient updating step. However, to keep this chapter in coherent order, the next revision of the QP solver which attempted to fix all of these problems will be discussed next.

7.2.2.2 Revision 2

Before the subproblem can be solved, the data for it must be generated. The generation step is computationally expensive yet performed solely on the PPE in Revision I.

Looking back at problem 7.1, it is apparent that the kernel matrix *A* needs to be generated before initiating the QP solver. While α_β, and y_β are easily generated by random indexing and copying from the corresponding global arrays, the kernel matrix *A,* and linear term *b* require many kernel computations. It should be pointed out that there is a shortcut for the calculation of the linear term once the global gradient is known. This shortcut is used for all but the first iteration. The standard method is:

(7.3) $$G_{\beta\delta}\alpha_\delta - 1_\delta .$$

The shortcut is:

(7.4) $$\nabla F_\beta(\alpha) - G_{\beta\beta}\alpha_\beta .$$

For details, see [64] and [35]. The significance of the shortcut is that it only requires working with *n* dimensions (the dimension of the subproblem) and that once the kernel is generated, no additional kernel iterations are necessary. This assumes that the global values of the gradient have been calculated, as they are in GPDT.

As mentioned in Chapter 3, the most expensive portion of the QP solver algorithm is the matrix multiplication used for finding the gradient of the subproblem objective function. There are two locations in which the multiplication is performed, differing only by the vector being multiplied. Because the size of the problem is selected by the user and kept unchanged, so is the kernel matrix. A proper strategy (as recommended in [35]) is to

divide the kernel matrix row-wise so that each SPE generates a portion of the output vector. In this implementation, all non intensive steps are performed on the PPE, only outsourcing the matrix multiplication step onto the SPEs. The QP solver, therefore, is rather trivial to implement.

The kernel matrix for a problem, once initialized, does not change for the duration of the running of the QP solver. This suggests that once the matrix contents are placed into the local stores of each SPE, they should remain there until the QP solver finishes. The task, therefore, is to efficiently place the rows of the matrix into the proper locations of each SPE. This task, however, is not that simple.

An important feature of every generated kernel matrix is that it is positive definite – meaning that the upper and lower triangular portions are mirror images of one another. In the original PGPDT source code, this fact is used to an advantage by only calculating the upper triangle of the matrix and then mirroring it across the diagonal, thus cutting kernel calculations by nearly half (the diagonal is calculated as well).

Recall that calculating an element $a_{i,j}$ of the kernel matrix requires the i^{th} and j^{th} input training vector, as well as the i^{th} and j^{th} output as shown below.

(7.5) $$a_{i,j} = y_i y_j K(x_i, x_j)$$

Because generating elements in the kernel, whether for subproblems, or for the gradient updating step described later, is very expensive, this task is a prime target for optimization. The details of this process will be postponed until the following sections. For the moment, assume that there exist utilities within the SPE module that perform this very step and that each training vector is assigned a unique id, or index value.

The second revision of the QP solver makes use of one SPE module, which is uploaded to all the SPEs, and performs subproblem data generation and matrix multiplication. A big challenge to overcome in this revision was to balance the workload among the SPEs during these two steps. If the kernel matrix was not positive definite (and thus symmetrical along the diagonal), this would be a trivial matter. Each SPE would be programmed to generate a unique, equally sized, portion of the matrix within its local store and, ignoring the differences in input vector sparsity, perfect workload balance between the SPEs would be achieved. This workload partitioning concept is shown in Fig. 7-4. Of course, this same balance can be achieved by ignoring the symmetry and just calculating each and every element, but this is unnecessary overhead. On the other hand, the advantage of this scheme is that the matrix multiplication step of the solver is perfectly balanced between the SPEs.

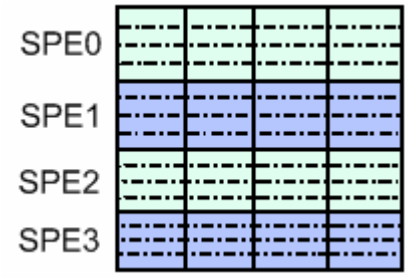


Figure 7-4: Calculating all kernel elements

On the other extreme, if only half the matrix is generated by distributing the workload as shown in Fig. 7-5, the kernel generation step is equally balanced. However, it becomes difficult to reflect the elements to form a complete matrix. Even if it was accomplished, the matrix multiplication step would be highly unbalanced.

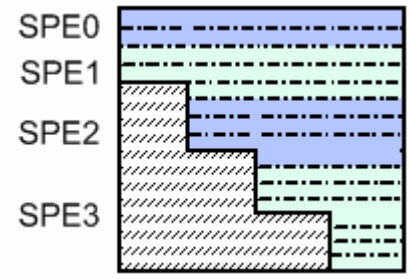


Figure 7-5: Equal balance of upper triangle, difficult reflection

The final solution that was implemented strikes a balance between the two extremes. The concept is to divide the kernel matrix into *N*x*N* blocks, each of which represents one job. The number of blocks to divide into depends on several factors, including the number of available SPEs and the size of the matrix. With this method, the number of kernel evaluations is just over half of the number of evaluations performed in the first case, and slightly over that of the second case.

During the initialization of the solver, each SPE is given a table of the blocks for which it is responsible for. Because the subproblem size does not change between iterations, this table is reused, saving on communication (similar to how the job information for each layer in the MLP algorithm was stored in a table on each SPE as described in Chapter 6). To enforce balance during the subproblem data generation step, each SPE is given about an equal number of blocks to process.

The first task is to figure out the size N of the length and height of each block, including the rightmost and bottom ones, the number of rows of blocks to be placed into each SPE. The limitation is the maximum block size (MAX_BLOCK_SIZE), which is set by a macro. The idea behind this algorithm is to find the minimum number of block rows per SPE such that the size of each block is within the maximum block size. The block size b_{size}, related to the number of block rows per SPE br_{spe} by equation 7.6.

(7.6) $$b_{size} = ceil_4\left(ceil\left(n \middle/ \left(NUM\_SPEs * br_{spe} \right) \right) \right)$$

in which n is the subproblem size and $ceil_4$ rounds up to the nearest multiple of four. Next, based on the number of blocks per column and row n_{bpcr} of the full kernel matrix, the number of block jobs is calculated and memory is allocated. The number of jobs is simply:

(7.7) $$T_{n_{bpcr}} = \sum_{k=1}^{n_{bpcr}} k \ .$$

Once the block jobs are allocated, initialization information for each of the NUM_SPE SPEs is generated and sent to the SPEs using the CMD_INIT command. The contents of this initialization structure are shown in Fig. 7-6.

General	SPE rank
Job Information	Block size Job list pointer Number of jobs Block rows per SPE
Matrix Multiplication	Start row Num rows Subproblem size Blocks per row/column
Gradient Updating	Total training vectors Sparse vector information Global gradient change
Kernel Parameters	Kernel Parameters Training vector dimensions

Figure 7-6: SPE-initialization structure

Each SPE is given an equal number of block jobs to generate (number of jobs should be roughly the same; Fig. 7-7). The actual jobs have not yet been set up as they require internal LS pointers of all the SPEs so that inter-SPE DMAs can occur. Recall that effective addresses can also map into the LS's of the SPEs. DMA transfers between SPEs utilize this effective address for direct SPE-to-SPE communication. Because of the dynamic allocation of memory, the local LS buffer addresses into the partial kernel matrices are not known until the SPEs are initialized (using the CMD_INIT command). The CMD_GET_KERNEL_LS command is sent after the CMD_INIT command which the SPEs respond to by returning their local LS address into these buffers. The system-wide effective addresses of the partial kernel buffers are obtained by adding the returned local LS addresses to the base LS address obtained when first setting up the SPEs.

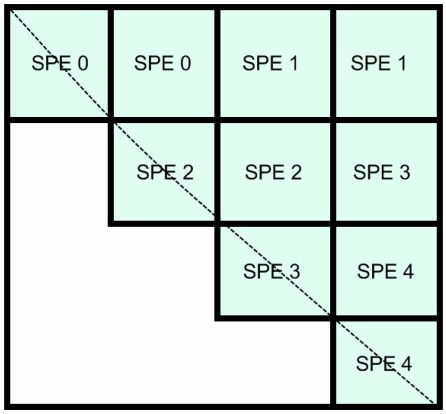


Figure 7-7: Block job SPE distribution

At this point, with the starting addresses of all partial kernel buffers in each of the SPEs known, the block jobs themselves are set up. The contents of a block job struct are shown in Fig. 7-8.

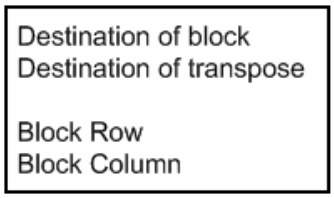


Figure 7-8: Job block structure

Each block is given a block row and column number which represents where in the full kernel matrix the block resides. Based on these two values and the previously obtained global effective addresses of each of the SPEs partial kernel buffers, the destinations of the block and its transpose are calculated.

So far, the number of kernel evaluations has been minimized, and each SPE is given an equal number of block generation tasks. The next goal is to enforce that at the end of the data generation step, each SPE holds an equal amount of consecutive matrix rows so that the matrix multiplication step is also balanced. This implies that the destination addresses of all the blocks during kernel generation should be equalized between the SPEs. There is no bind between a block job and the SPE at which it was generated. Blocks generated on one SPE may end up on any of the other active SPEs depending on their destination addresses. In fact, most of the time, the destination address of the block and that of its transpose belong to two different SPEs (Fig. 7-9). The destined SPE is chosen based on the number of block rows per SPE and the row number of the block (and column number of the transpose of the block).

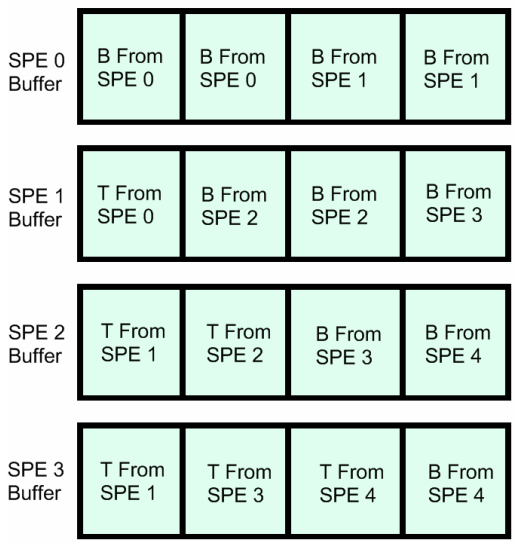


Figure 7-9: Final locations of all the blocks (B: Block, T: Transpose)

Once all block jobs are created, the CMD_JOBS_READY command is sent to each SPE, at which point the SPEs DMA in their personal job lists from main memory into their LSs.

At each iteration of the main loop, there is a subproblem that needs to be solved. The PPE is responsible for choosing those indices from the global training set that will make up this subproblem based on previous iterations. Once, chosen, the PPE generates several arrays in memory that will be used by the SPEs, including the DMA lists for the sparse training vectors in the current subproblem. The pointers to this data are encapsulated into a *problem_data* structure shown in Fig. 7-10.

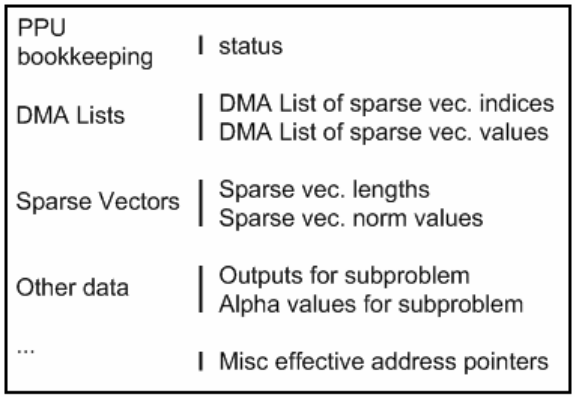


Figure 7-10: Problem data structure

Next, the CMD_GENERATE_KERNEL command is sent to all SPEs along with the effective address of the data generated above. The SPEs begin generating their portions of the kernel matrix for this subproblem. The block generation algorithm is shown in Listing 25. Generation and sending of blocks is intertwined by using multi-buffering. Increasing the number of buffers allows for having more blocks (and their transposes) in

transit at any one time. In software, the buffers are encapsulated into job contexts, which are accessed sequentially while processing block jobs. Having four job contexts, each with their own buffers and DMA transfer statuses, showed to have the best size vs. speed performance.

```
DMA in data given in the problem data structure

For all block jobs assigned to this SPE
        Select next block job context (round robin)
        Wait for DMA transfer completions on this context
        Clear block and block's transpose buffer memory
        Generate the block
        Begin DMA transfer of block to its final destination
        Generate transpose of block
        Begin DMA transfer of transpose to its final destination
End Loop
```

Listing 25: Block generation algorithm

The actual task for generating each block has also been heavily optimized. Recall that in order to generate the $(i,j)^{th}$ element in the kernel matrix, the i^{th} and j^{th} sparse training vectors are needed. Simplified pseudo code is shown in Listing 26.

```
While there are rows remaining to be generated
        DMAL 4 sparse input vectors (four rows)
        Expand the sparse vectors into non-sparse format
        While there are columns remaining
                DMAL up to 16 sparse input vectors (columns)
                Use optimized kernel generation module to
                  generate four rows of the kernel block
        Loop
Loop
```

Listing 26: Generating the actual block

The function generating the transpose has been optimized as well. It involves a loop in which a sequence of vector instructions are used to generate index values into the source block and destination transpose block. Once the source and destination pointers are obtained, simple vector shuffling instructions are used to copy the data. The loop is designed to process a four by four block on every iteration.

An additional step of subproblem data generation is creating the linear term of the Lagrange function. As mentioned earlier in this section, there is a shortcut to creating this array (Eq. 7.4). Each SPE generates a portion of the linear term based on the rows of the kernel that are in its memory and DMA's it into main memory (effective address is stored in the initialization data obtained along with the CMD_INIT command). To initiate this step, the PPE must wait for an incoming signal from each SPE after issuing the CMD_GENERATE_PKERENL command. This synchronization step acts as a barrier, ensuring that the final subproblem kernel is ready and distributed among the SPEs. At this point, the CMD_GENERATE_PLINTERM command is sent. The SPE simply

DMAs in the data it needs and performs a matrix-vector multiplication corresponding to Eq. 7.4.

Having the data generated, the subproblem solver can be run. To shorten development time, the revision 1 solver code was ported to run on the PPE by utilizing the spu2vmx and vec_types.h header files. Doing so enforced the use of the PPE's Altivec unit for vector operations. The code was further modified, replacing the two expensive matrix multiplications with interface functions into the SPEs.

The two matrix multiplications that are performed differ only by the vector being multiplied. Because each matrix multiplication operation requires the effective address of the input vector and output vector, these addresses are given to the SPE along with the CMD_INIT command inside the initialization structure (Fig. 7-10). These pointers are kept until the subproblem is solved. The correct choice for the input and output effective addresses is chosen based on a parameter in the CMD_PERFORM_MULT command, which is shown in Fig. 7-11.

Multiply Command (8 bits)	Set (24 bits)

Figure 7-11: Multiply command

The parameter *Set* is used to select the corresponding effective address for the source of the vector multiplied and destination of the product vector. Because the QP solver runs over many iterations, requiring only one mailbox message per multiplication reduces communication overhead.

7.2.3 Kernel Element Generation

In this work, only the Gaussian kernel was implemented due to lack of time. The Gaussian kernel, however, is the most common and more than sufficient for the purposes of this work. The time it would take to implement the other kernels was better spent in optimizations so that the potential of the Cell Processor could be fully exhibited.

In general (ignoring the working set method) the Gaussian kernel matrix elements are generated using:

$$\alpha_{i,j} = \exp\left(-\left(norm_i + norm_j - 2.0 * \left\langle vec_i, vec_j \right\rangle\right) * \sigma\right) \tag{7.8}$$

in which i and j are two training vector indices and $a_{i,j}$ is the element in the i^{th} row and j^{th} column (as well as j^{th} row and i^{th} column due to symmetry). Note that when generating kernel matrices for subproblems, the i's and j's take on values from the subset of the indices in the current working set.

As mentioned previously, besides caching, the optimization of the kernel generation code is without question one of the greatest improvements that can be done to speed up the GPDT algorithm. In that regard, an SPE optimized function was written designed to efficiently calculate several elements of a single row of the kernel matrix at a time. The function signature is shown in Listing 27.

```
void GetKgaussKernelRow(  vec_float4 *vec_output,
                          vec_float4 *vec_exp_row, float exp_row_nor,
                          vec_float4 *vec_input_rows, int *vix, int *vlx, float *sparse_nors, int num_ell )
```

vec_output:	Pointer to result vector.
vec_exp_row:	The main input training vector (non-sparse).
exp_row_nor:	The norm value for the row above (pre-calculated during initialization).
vec_input_rows:	All the input training vectors (in sparse form) which will be evaluated against the main input training vector using the kernel.
vix:	Sparse indices for the non-zero values for each of the training vectors above.
vlx:	Sparse lengths for all the training vectors above.
sparse_nors:	The norms for all the sparse input rows.
num_ell:	The number of the non-main sparse training vectors (the number of kernel elements that will be calculated). This number must be a multiple of four.

Listing 27: Kernel row generation function signature

Note that all the arrays corresponding to the sparse input rows vary in length (the *vlx* array has information about all their lengths) and each array is contiguous in memory.

Using function 7.8 as a reference, it is clear that the dot product is the first step performed for any pair of input training vectors. In the *GetKgaussKernelRow*, the dot product is performed for four sparse training vectors at a time, each paired with the main non-sparse vector. For every four sparse vectors (outer loop), the inner loop performing the dot product executes a number of times equal to the longest sparse vector. Once the loop exceeds any of the four vector lengths, the dot product at that iteration for that element is calculated but not added to the running sum (dot product is a running sum of products for each element in the vector). This is done by using the SPE vector-wise compare and select instructions as demonstrated in section 5.4.1.2.

To better explain the procedure, a toy example in which four elements are to be calculated follows. *Vec_exp_row* is the main non-sparse vector against which the four sparse vectors in *vec_input_rows* are processed. The lengths of the sparse rows are 2, 1, 3, 1 in that order.

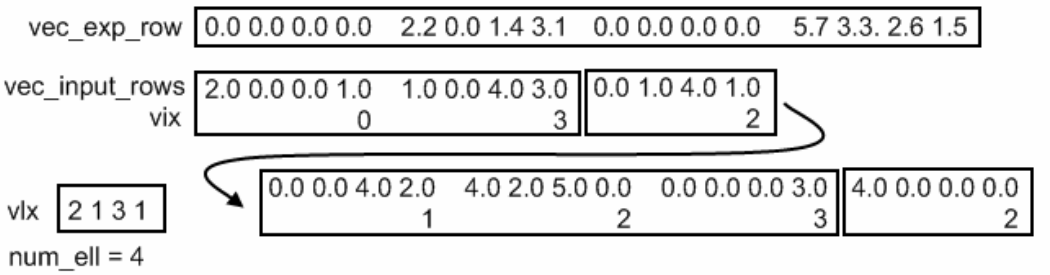


Figure 7-12: Parameters into GetKgaussKernelRow function

Because there are four elements, there is only one iteration of the outer loop. During this loop, the following pointers will be assigned:

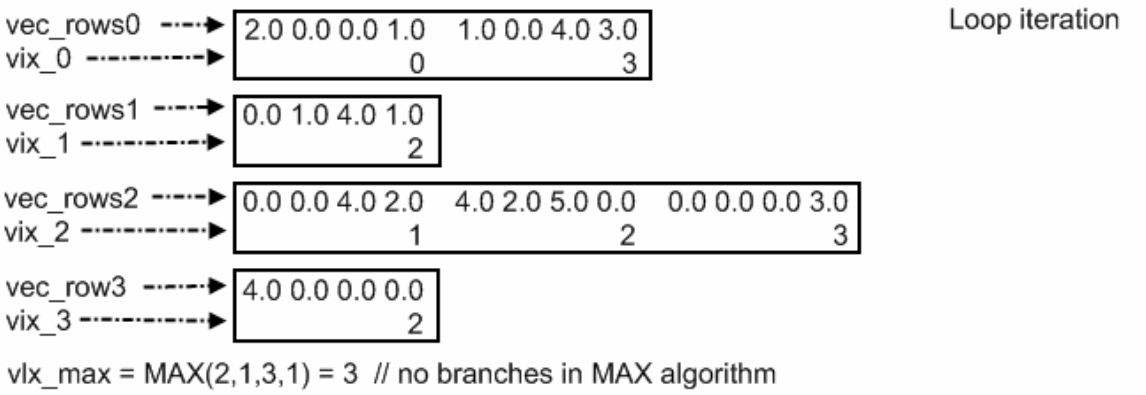


Figure 7-13: Pointers assigned at first iteration of loop

Next the inner loop (*jj* is the loop control variable) starts and iterates 4 times (MAX(2,1,3,1) = 3 → *jj* = 0,1,2,3). All the *vec_add_[n]* values are calculated whether the results will be added to the main dot product vector or not (see next step). Each of the dot product vectors *vec_add_[n]* are summed across and inserted into element positions 0, 1, 2, and 3 of a temporary vector.

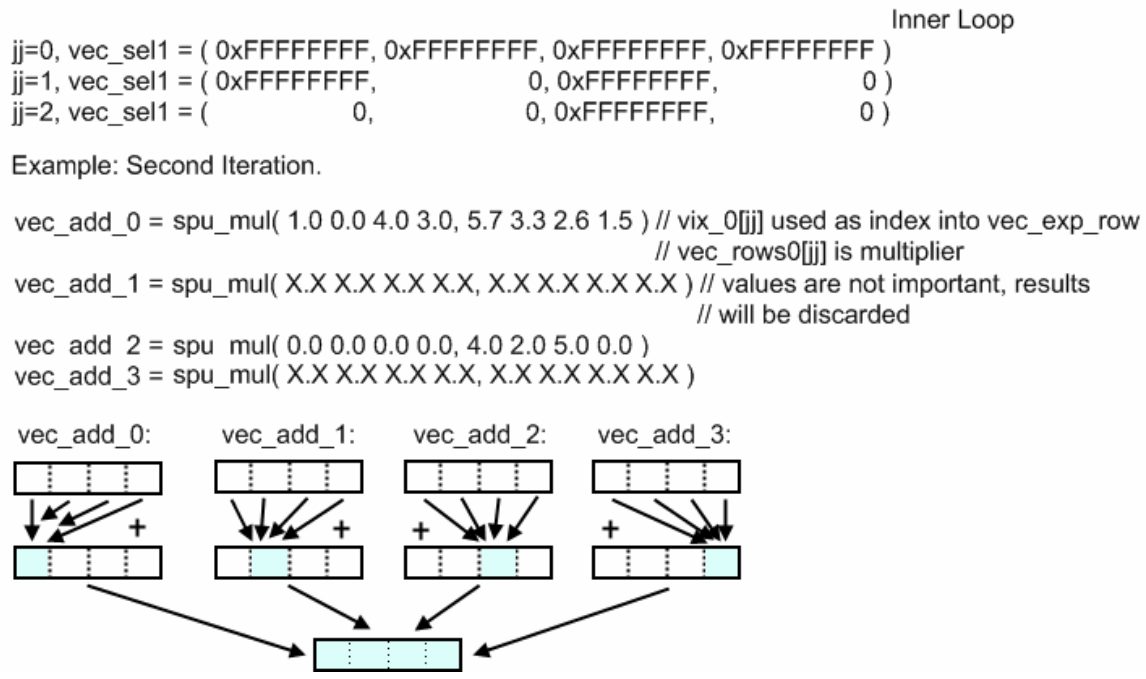


Figure 7-14: Updating of the running dot product

Because it's very likely that all the sparse vectors are not the same length, at every iteration, the loop counter *jj* is made into a vector by copying the value four times. The resulting *jj* vector is compared to the *vlx_max* vector. A select mask vector is formed that is used to modify the contents of the temporary vector created in the previous step. Those elements for which *vlx* is less than *jj* are set to zero (using the select masks created and

the vector select instruction). The adjusted temporary vector is added to the current position in the main dot product array.

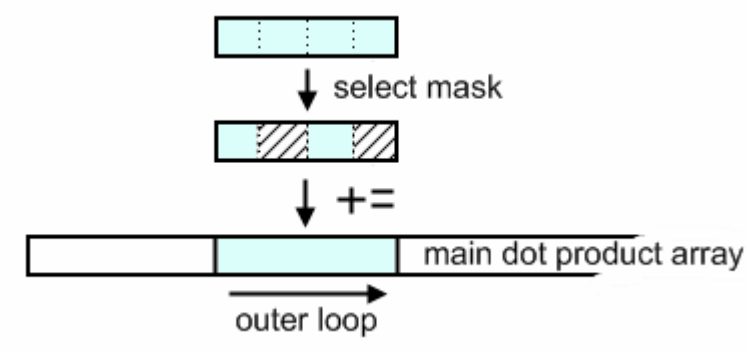


Figure 7-15: Adding the result to the running dot product array

Finally, once the inner loop is finished, the rest of the operations in function 7.8 can be performed as simple aligned vector operations on the current location of the main dot product array.

7.2.4 Updating the Gradient

Because existing GPL code was available, it was relatively easy to optimize one section of the algorithm while keeping the rest unchanged. For example, the original gradient updating code was used throughout the development of revision 1 of the QP solver. A selection of small training problems was used for verification that the obtained results did not change from the original, unmodified code.

The gradient updating portion of the algorithm is most computationally expensive when the number of total training vectors becomes large compared to the number of training vectors in a single working set. The task boils down to a large matrix-vector multiplication. The problem is that most of the contents of this matrix are not in memory and must be calculated. The caching technique in the PGPDT implementation is very useful here, but only up to some size of the training set. Nevertheless, it was updated to work with the Cell implementation and could be turned on and off as seen fit.

As a review, the formula to calculate the gradient of the dual Lagrange objective function is:

$$\nabla F\left(\alpha^{k+1}\right) = \nabla F\left(\alpha^{k}\right) + \begin{bmatrix} G_{\beta\beta} \\ G_{\delta\beta} \end{bmatrix} \left(\alpha_{\beta}^{k+1} - \alpha_{\beta}^{k}\right). \tag{7.9}$$

Note that the columns of the matrix $\begin{bmatrix} G_{\beta\beta} \\ G_{\delta\beta} \end{bmatrix}$ are the sparse columns from the global kernel matrix of column indices corresponding to the training vector indices of the current working set.

To reduce the number of calculations, the PPE obtains the value $\left(\alpha_\beta^{k+1} - \alpha_\beta^k\right)$ and discards those indices for which the result is less than a small value (DELTAsv). The remaining set of indices is evenly distributed for processing among the available SPEs. Each SPE produces a partial sum of the $\begin{bmatrix} G_{\beta\beta} \\ G_{\delta\beta} \end{bmatrix}\left(\alpha_\beta^{k+1} - \alpha_\beta^k\right)$ value that is added to the global sum by the PPE.

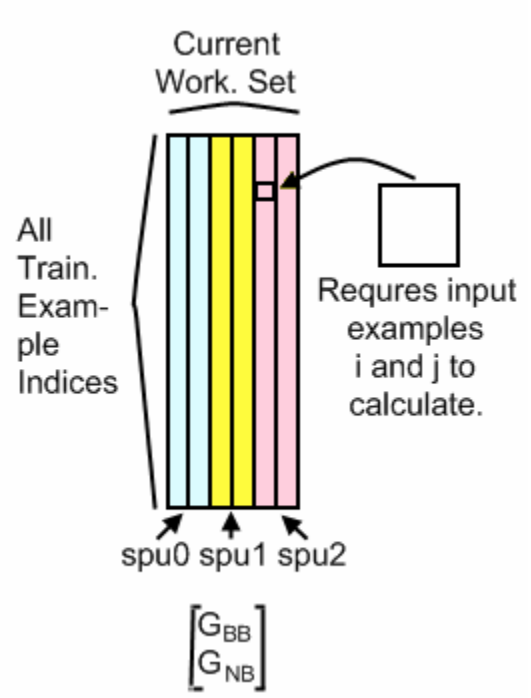


Figure 7-16: Kernel Matrix generation job distribution among SPEs

In the first revision, the gradient updating code was placed into a separate SPE software module and ran on different SPEs than the QP solver (five gradient updating SPEs and one QP solver SPE). Once revision 2 of the QP solver module was finished, however, the gradient updating module was merged with it to produce one SPE module capable of performing both procedures. While this made the code size slightly larger, there was some code which was shared including the DMA handling and kernel element generation. Merging the two modules also made it needless to perform expensive context switching and allowed for the maximizing the utilization of all six available SPEs. The increased complexity came from the management of memory (pointer handling) since the usage of the LS space between the two was intentionally overlapped for higher efficiency.

The first step that needs to occur after uploading the SPE module to the SPEs is the sending of initialization information. This information includes any kernel parameters, the number of total training vectors, dimension of the training vectors, and effective addresses of various training vector data (sparse lengths, sparse indices, sparse values, norm values). Once the information is received, the LS memory is organized by setting

various pointers for both the gradient updating step and QP solver step. For both cases, any leftover memory is maximized for critical buffer usage such as double buffering. This way, the LS space is optimized.

Performing gradient updating follows the same PPE-SPE programming model as the QP solver. However, extra information needs to be sent to the SPEs at each step. Three mailbox messages are sent to all the SPEs as shown in Fig. 7-17.

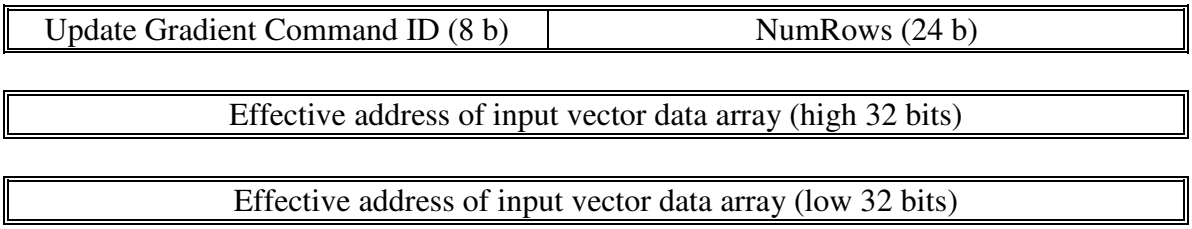


Figure 7-17: Sequence of mailbox messages sent to SPEs at gradient updating step

Note that the *NumRows* and effective address is different for all SPEs causing each SPE to DMA different information (data parallelism).

The SPE uses the received effective address to download the array of *NumRows* input vector information from main memory that it is responsible for. Table 5 shows the contents of a single instance of the vector information struct.

Name	**Type**	**Description**
vec_id	*uint32_t*	The global index of the training vector
sparse_vlen	*uint32_t*	the sparse length of the vector
norm	*float*	the vector norm
grad	*float*	the value $\left(\alpha_\beta^{k+1} - \alpha_\beta^k\right)$
EA_row_start	*addr64*	the memory location of the sparse vector
EA_row_idxs	*add64*	the memory location of the sparse vector indices
KernelRowInCache	*char*	is the kernel row in the cache
CacheOn	*char*	should the cache be used/updated
EA_kernel_row	*addr64*	if the cache is on, the memory location of the kernel row

Table 5: Vector information struct

The pseudo-code in Listing 28 shows a very simplified and non-optimized version of the procedure. The text following gives an overview and comments on each step.

```
training_vec_remaining = total_training_vectors (matrix rows)
While training_vec_remaining > 0 (Loop A)
        ws_size = MIN(MAX_WORKING_SET, training_vec_remaining)
        DMA in the next ws_size norm values (norm) and sparse vector lengths (vlx) for this set of
          training vectors
        ws_size_remaining = ws_size
        While ws_size_remaining > 0 (Loop B)
                inner_ws_size = maximum sparse vectors that can fit into the buffer on the LS
                DMA in the next inner_ws_size input vectors (sparse values and indices) from the
                  current working set (Call this set r_iws)
                For all rows (r) assigned to this SPE (Loop C)
                        If this kernel row is in the cache in main memory
                                DMA in the current portion of the kernel row (size=inner_ws_size)
                        Else, need to generate it:
                                DMA in the sparse input vector values and sparse indices for this
                                  row
                                Expand the sparse row into non-sparse format
                        Calculate the kernel values that can be obtained from the current row r an
                          all sparse rows in the inner working set (r_iws)
                        If cache is enabled
                                DMA the portion of the kernel row back into PPE cache
                        Update the delta gradient (st_out) for this section of the current working set
                Loop (C)
                ws_size_remaining = ws_size_remaining – inner_ws_size
        Loop (B)
        Send the final partial gradient to the PPU
        training_vec_remaining = training_vec_remaining – ws_size
Loop (A)
```

Listing 28: Pseudo-code of gradient updating on the SPEs

The total training vectors, as the name implies, is the total number of training vectors in the full set. Because this number can become very large, only a portion of the total training vectors are worked with at any time. The outer most loop (*Loop A*) is responsible for sequencing over the entire set in chunks that won't overwhelm the LS. The maximum working set size for each iteration is MAX_WORKING_SET. Two partial arrays of data are downloaded from main memory: *vlx* – the sparse lengths of each training vector, and *norm* – the norm of each of the training vectors in the current portion (or working set).

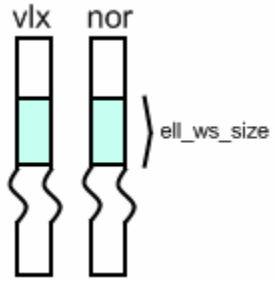


Figure 7-18: The contents of the vlx and nor vectors received by the SPE in Loop A

The next inner loop (*Loop B*) is responsible for downloading as many sparse vectors from the working set as possible into the available buffer space. The *vlx* array, which holds the

sparse length of the vectors in the current working set is used to figure out this number, knowing that each sparse entry in the sparse vectors requires twenty bytes (16 byte 4-element vector and 4 byte sparse vector index).

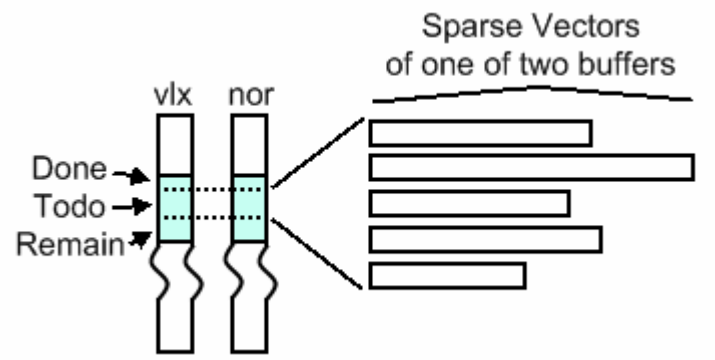


Figure 7-19: The actual sparse vector contents received within Loop B

The two loops so far execute identically on all SPEs since the set of all global training vectors is common between them. The next inner loop (*Loop C*), however, iterates over all the training vectors indices that were assigned for that SPE. When generating the row information for the SPEs, the PPE sets a flag in the information struct whether that row is currently in the cache. The SPE checks this flag here and if it is, downloads the current portion of the row (the portion is determined by *Loop B*'s current training vector index and *inner_ws_size*). If the flag is not set, the contents of the current training vector (sparse values and indices) are downloaded and expanded into non-sparse form. The expanded training vector array is used as the second parameter of the *GetKgaussKernelRow* function. The remaining parameters include the current set of sparse training vectors downloaded by the first inner loop, the corresponding *nor* and *vlx* values downloaded in the outer loop, and the number of elements to generate is equal to *inner_ws_size*.

Once this portion of the kernel row is generated, column based matrix vector multiplication is used to obtain the partial change of the gradient due to this row. In other words, all elements in the resulting kernel row portion are multiplied by the $\left(\alpha_\beta^{k+1} - \alpha_\beta^k\right)$ value associated with the current row of the inner-most loop. This is easily done by copying this value into a four-element vector and performing a set of SIMD operations. The partial change of the gradient is added to the proper location of *st_out*, which is the final partial change of the gradient for this SPE of size equal to *ws_size* (determined in *Loop A*).

Fig. 7-20 shows a summary of the order in which the loops generate the matrix and produce the change in the gradient.

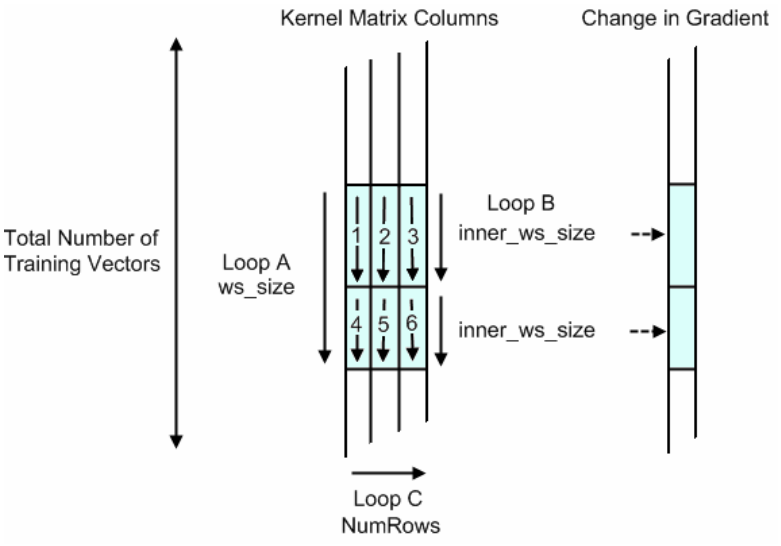


Figure 7-20: Updating a portion of the product vector by a single SPE

In the first revision of the gradient updating code, double buffering was performed in Loop B over the contents of the sparse vector values and indices as shown in Fig. 7-21 and simplified pseudo-code in Listing 29.

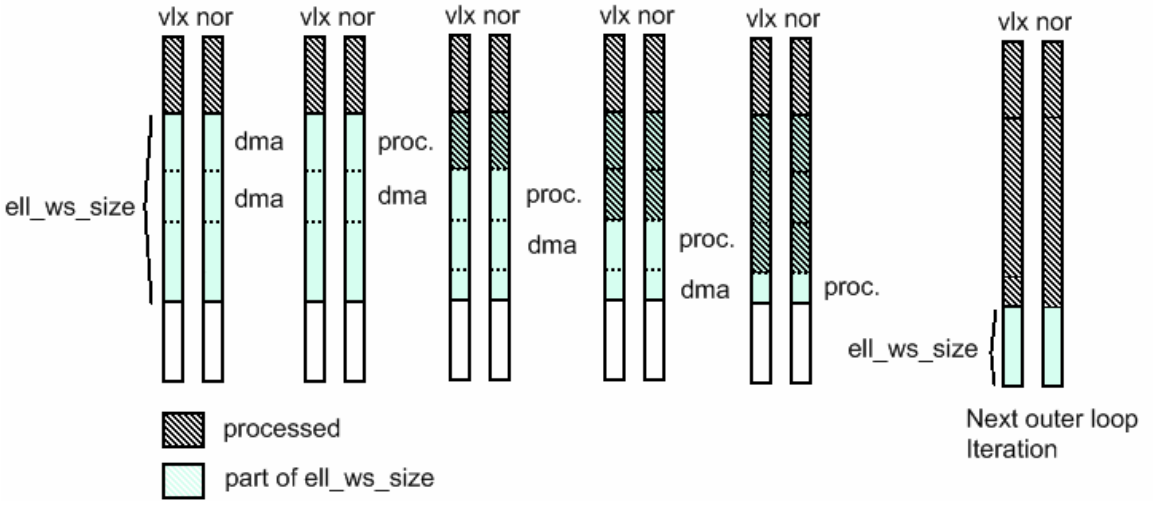


Figure 7-21: Double buffering in the first revision of the gradient updating SPE module

```
training_vec_remaining = total_training_vectors
While training_vec_remaining > 0 (Loop A)
        ws_size = MIN(MAX_WORKING_SET, training_vec_remaining)
        DMA in the next ws_size norm values (norm) and sparse vector lengths (vlx) for this set of
          training vectors
        ws_size_remaining = ws_size
        inner_ws_size_0 = max that can fit into buffer 0
        inner_ws_size_1 = max that can fit into buffer 1
        Start DMA of sparse vector data into Buffer 0 (inner_ws_size_0) and Buffer 1
          (inner_ws_size_1)
        While ws_size_remaining > 0 (Loop B)
                wait for Buffer 0 DMA to complete
                Loop C for Buffer 0
                Start DMA of next inner_ws_set into Buffer 0
                wait for Buffer 1 DMA to complete
                Loop C for Buffer 1
                Start DMA of next inner_ws_set into Buffer 1
        Loop (B)
        Send the final partial gradient to the PPU
        training_vec_remaining = training_vec_remaining – ws_size
Loop (A)
```

Listing 29: Double buffering in the first revision of the gradient updating SPE module

It was soon discovered that double buffering at this level provided very little advantage as most of the DMA stalls occurred in *Loop C* when downloading the sparse row contents (or partial kernel row) for each row index assigned for the SPE. The second revision of the gradient updating code, therefore, removed double buffering in *Loop B* and implemented multi buffering in *Loop C*. Fig. 7-22 graphically shows a toy example of a double-buffering setup in *Loop C*.

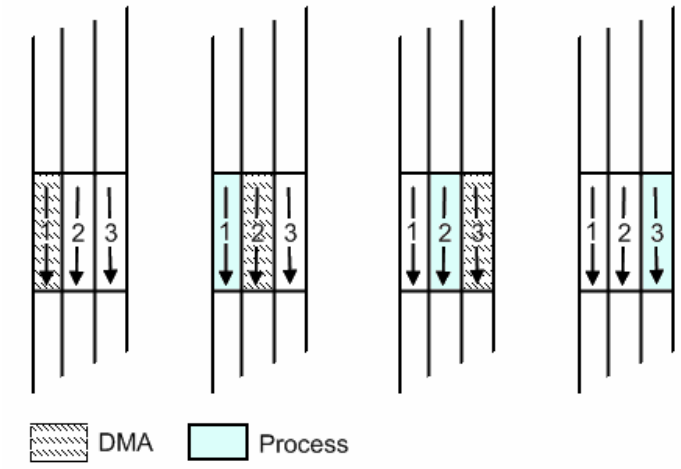


Figure 7-22: Double buffering in second revision of gradient updating SPE module

7.3 Cascade SVM

As introduced in Chapter 3, the cascade SVM achieves its level of parallelism by generating multiple asynchronous and independent QP problems. Each problem's data is generated from a certain subset of the full set of training vectors. By analyzing the α values obtained by solving each QP problem, the support vectors for that QP training subset can be deduced and combined with support vectors from other solutions to form a new training set for the next problem in the tree. The output set of support vectors is a subset of the input training vectors originally chosen for a particular problem, and therefore the output is always smaller than or equal to the input. There is no guarantee, however, that combining the outputs from two or more solvers will produce an input small enough to fit on the available LS space for a single SPE solver. The problem, therefore, is to guarantee that the number of training vectors for any new problem does not exceed a fixed maximum amount.

In this implementation, the Cascade SVM method was implemented to function as a filter into the GPDT solver. In other words, the Cascade SVM was not capable of solving the problem all by itself.

7.3.1 Dependency Tree and Job Queue

In the implementation, the PPE keeps track of a dynamic dependency tree for all the currently active QP solvers. Active solvers may or may not be running due to dependencies. The dependency tree is formed from individual smaller trees called *Cascade421*'s. Each *Cascade421* has three layers with a total of seven elements as shown in Fig. 7-23.

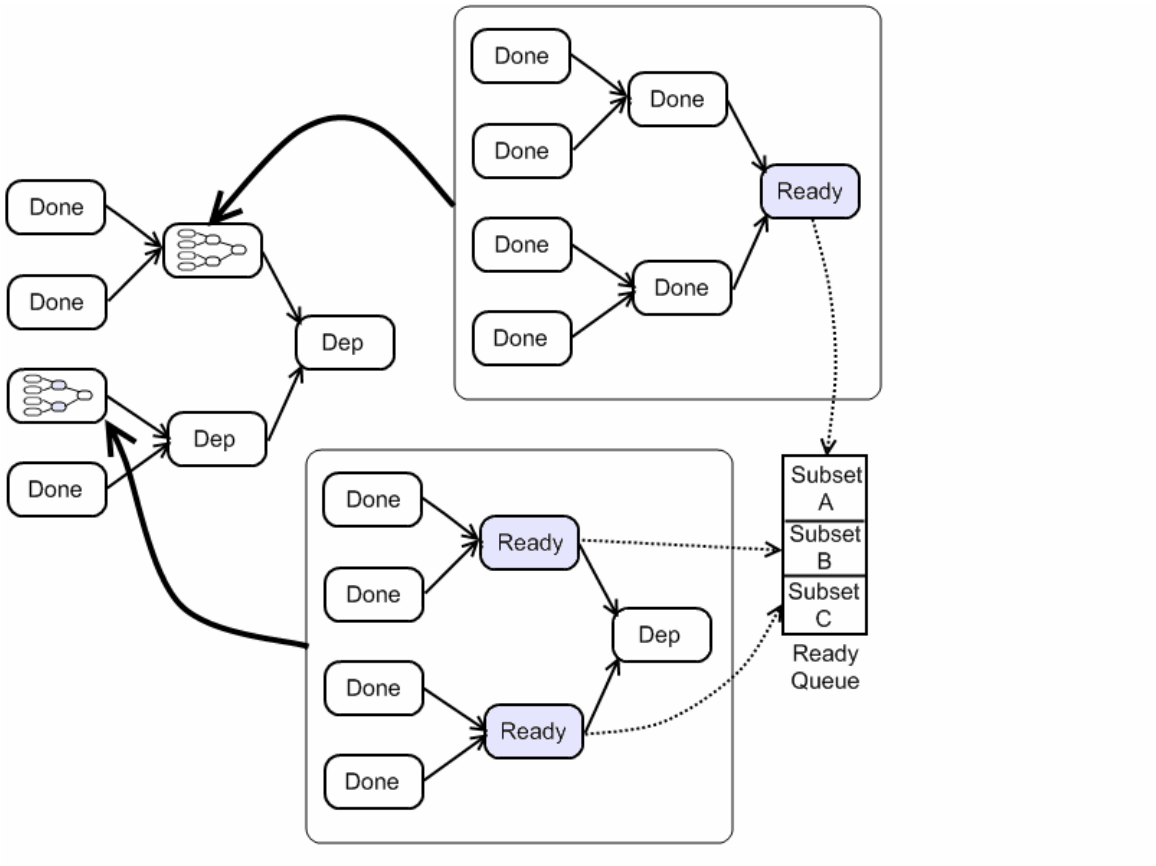


Figure 7-23: Dependency tree of Cascade421 objects

Before proceeding, the reader is reminded that each training vector is indexed by a unique index value. A subset of training vectors is, therefore, represented as a subset of indices.

In the figure, solid arrows show the dependencies between the elements. Each element holds a set of training indices. If the set of indices is too large for a single solver, the element spawns (farms out) a new *Cascade421* element which automatically divides the set into four subsets for its first layer. This process effectively creates new dependencies that must be resolved before that element can proceed. Any of the seven elements in a *Cascade421* object can spawn a child *Cascade421* object, hence creating a "tree of trees" structure. If the number of indices is below the maximum, a *TrainingIndices* object is created holding the indices of that element. The *TrainingIndices* object is queued in an independent job queue for processing (also shown in the Fig. 7-23) and the element is marked as ready. Whether a node farms out its training indices or not is transparent to the parent *Cascade421* object as it simply waits until that node is marked "done."

Each time an element is marked ready for processing (that is, the number of training indices was less than the maximum allowed) and the given training set is placed on the ready queue, the element holds a pointer to that set, as shown in the figure using dotted arrows. The PPE effectively alternates between processing the dependency tree and processing the ready queue. Following the ready queue processing, at the next

dependency processing, it will check if the problem has been solved and change its status accordingly.

Elements in a *Cascade421* are numbered 0-6 starting at the left layer, moving from top to bottom. In Fig. 7-23, elements 2 and 4 are farmed out from the topmost *Cascade421* object. The status of the tree represents the time after the tree has been processed for dependency resolutions but before processing the job queue. Element 6 is ready, which implies that its indices are in the job queue in the form of a TrainingIndices object and are ready to be processed.

The reason for choosing the *Cascade421* representation is that it could easily generalize to the improved cascade SVM or M^3-SVM, as described in Chapter 3.

As mentioned above, the ready queue holds instances of *TrainingIndices* objects, and not actual problems. A single iteration of the dependency tree could create thousands of new ready elements. If the problem was generated before being queued, all the memory taken up would defeat the purpose of this divide and conquer technique. Instead, the problems are generated as the *TrainingIndices* objects are dequeued during the job queue processing stage as will be described later.

7.3.2 Desired Filter Ratio Parameter

With the addition of the Cascade SVM implementation came a new parameter – the desired filter ratio (-R) – which had the effect of controlling the amount of filtering performed by this step. The *Cascade421* element keeps track of the number of support vectors coming into and leaving each layer. Fig. 7-24 shows the locations at which these numbers are calculated. When all layer 1 solvers are finished, the ratio *b/a* is calculated. When all of layer 2 is finished, the ratio *c/b* is calculated. If either one of these ratios is greater than the value passed in using the -R parameter, the *Cascade421* element will finish early, and notify its parent *Cascade421* object. A high desired filter ratio will force more *Cascade421* elements to run to completion. A lower value will cause the *Cascade421* elements to quit early due to loss of filtering efficiency (large ratio).

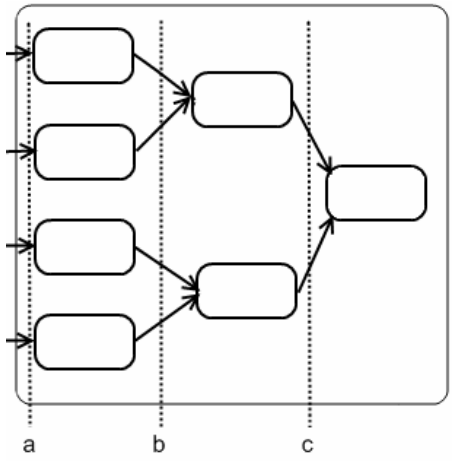


Figure 7-24: Points in the Cascade421 element at which the number of support vectors are checked

7.3.3 Improved QP Problem Solver

Although the first revision of the QP solver was completely replaced by the second revision in the Cell GPDT implementation, it was an excellent starting point for designing the problem solver for the Cascade SVM implementation. Here are the problems that were inherent to the first revision of the problem solver in the context of the GPDT implementation:

- Overhead due to copying of data into "DMA'able" (aligned, padded) arrays
- A maximal subproblem size of 200
- No potential for parallelization
- No subproblem data generation

The overhead for the first problem is now a necessary step rather than overhead. In other words, the problem does not exist in memory beforehand, and is generated at the time when it is dequeued. This implies the generation of all DMA'able arrays at that time. There is no conversion being done as was the case of the GPDT implementation. Furthermore, the generation of the DMA'able arrays is built into the new PPE side of the solver and is executed in a pipelined manner. Basically, the PPE readies arrays for the next problem in the queue while existing problems are being solved on the SPEs. So, not only is this not considered overhead, but it can be "hidden" from the total execution time.

The problem size limitation, unfortunately still exists in the design of the cascade SVM solver.

In the GPDT implementation, there was no potential of parallelization due to the linear and sequential problems. Here, the queue often holds hundreds of TrainingIndices which can be transformed into independent problems. After all, the main idea behind the Cascade SVM is the generation of multiple parallelizable problems.

The last problem in the list was solved by devising a clever method for generating the subproblem data on the SPE. Granted, this was not really a problem, just a lack of a feature that turned out to be feasible.

The problem solver (named *SPQProblem*), just as most of the code, was written using C and C++ and currently consists of two main components. The PPE component functions as an interface for assigning and processing problems and controls the execution of the SPE component (an SPE module). An earlier version of the solver, before the new problem generation technique was devised, included a second SPE module, for a total of three components. The task of the second SPE module (generator module) was to generate the data for the problem and send it directly to the solver SPE module. The generator SPE and the single solver SPE were tied using quite a complex communication scheme and worked in a pipelined fashion. The number of SPE generator modules and the selection and amount of work done on each was adjustable. Two architectures were tested: three solvers having one generator each (Fig. 7-25a), and two solvers having two generators each (Fig. 7-25b).

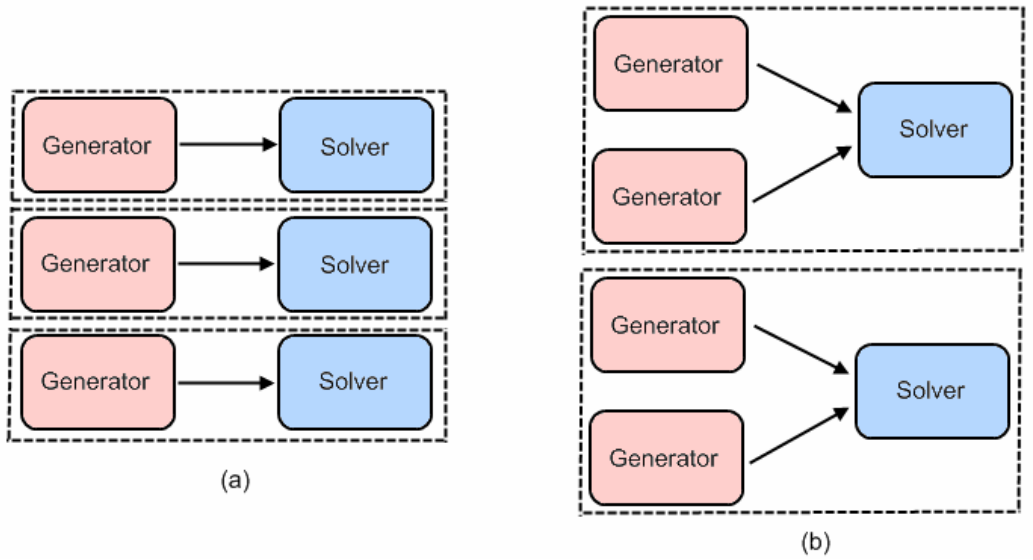


Figure 7-25: Pipelined QP solver SPE modules

Eventually, this concept turned out to be quite unnecessary and had an unresolved bug due to a race condition between the SPEs. Results were therefore not collected with this method. The realization of a new technique that performed generation of the problem data within the solver module lead to the abandoning of the overly complex pipelined version. The rest of this section refers to the new concept, although the interested reader is invited to examine the source code, which is still present as the *SPQProblem* class was written to be versatile enough to support both methods.

The *SPQProblem* class takes care of reserving and initializing an SPE before use. Being that there is one SPE module per solver, the number of *SPQProblem* objects created is equal the number of available SPEs for maximum parallelization. Initializing the SPE modules follows the same pattern as elsewhere in this work. Namely, an initialization command is sent, followed by the high and low 32 b portions of the effective address of the initialization structure located in main memory. The SPE downloads and stores the structure which contains information about the kernel parameters and dimension of the training vectors. The PPE module initialization consists of having various pointers assigned to the global problem data so that sub problems can be generated given a *TrainingIndices* object as a parameter. The data includes the global Lagrange multipliers α's, training vector output values *y*, and the *sKernel* object which holds the actual training vector data.

The *SPQProblem* class was designed to be controlled asynchronously and contains a two-element pipeline – one for allocating memory and gathering the required information from the global data, and one for running the actual solver. This model makes it easier to interface with the SPEs in the job queue handling code (Listing 30).

```
Do
        Loop over all active SPQProblem solvers
                Process SPQProblem solver
                If SPQProblem solver is ready for new input
                        dequeue the next problem from the job queue
                        initialize the dequeued problem and use AssignProblemData to
                          assign the problem to the solver
                        Process SPQProblem solver
While there are more problems in the job queue
```

Listing 30: Job queue handling

Two interfaces into the *SPQProblem* are *AssignProblem* and *Process*. Assigning a problem involves creating an internal context representing the problem and placing it into the internal pipeline. The SPE command data structure is generated as well. It is sent to the SPE in the next stage of the pipeline. *AssignProblem* takes a *TrainingIndices* instance as a parameter and uses it to selectively copy values from the global problem arrays. Fig. 7-26 shows the idea.

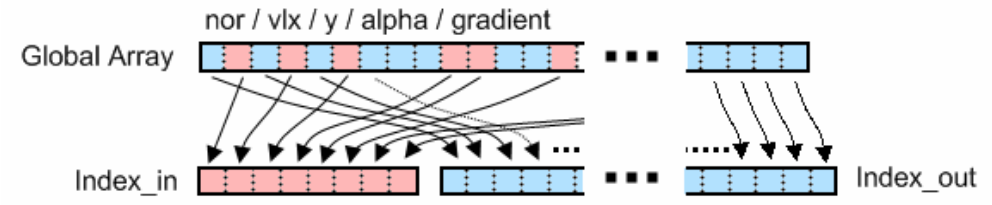


Figure 7-26: PPE AssignProblem random indexing of global problem arrays

The core interface function is *Process*, which manages the internal problem contexts by pushing them along the pipeline. Each time the Process function was called, the PPE updated the status of the currently active problem contexts in its pipeline. The problem contexts could be in any one of the three slots – init, gen, or solver – at a time.

When a problem context enters the solver slot, three commands are sent to the SPE, following the usual pattern:

New Problem Command	PPE Rank (=0)

Effective address of problem command data (high 32 bits)

Effective address of problem command data (low 32 bits)

Figure 7-27: Commands send to SPE when starting QP solver

The parameter of the first command is the source rank. It is used so that the receiving SPE knows what processing element the command came from. Each SPE has a table of information about the other processing elements that is indexed by their unique rank number.

The pseudo-code in Listing 31 gives an overview of the operations that occur on the SPE in the most recent version.

```
Loop Indefinitely
        Wait for command
        Case: Received new problem command
        DMA in the problem information structure
        DMA in the y output array for this problem
        Generate the kernel
        DMA in the α array for this problem
        If using the standard linear term (vector of -1's)
                Generate the standard linear term
        Else if generating the linear term locally
                Generate the linear term
        Run main solver loop
        Place updated <alphas> back into main memory
        Signal PPE
Repeat
```

Listing 31: SPE solver pseudo-code

The problem information structure is downloaded from main memory. It contains effective addresses of various data that will be needed and where the results are to be placed in main memory, the problem id, the size of the training set, a number of solver options, the quadratic optimization problem parameters, etc.

The generation of the kernel matrix is one of the new items added for the cascade SVM implementation. The procedure is explained in the following section. Once the matrix is generated, the rest of the data that is required for solving the problem is DMA'ed in. In the case of the cascade SVM, the linear term is simply a vector of -1's. When done, the resulting Lagrange Multipliers are uploaded back into main memory and the PPE is notified.

On the next call to the *Process* function, the problem is flushed out of the pipeline and the next problem is sent to the SPE.

7.3.3.1 Kernel Generation

Only the minimal amount of data is transferred onto the LS for the generation of the kernel matrix. Since the matrix will be positive definite, only the upper triangle of the matrix is generated. However, it is not a perfect triangular matrix. To best utilize the SIMD units on the SPE it is required to work with four values at a time. In effect, 4x4 blocks are generated at a time. The data is placed into memory in a compressed form so as to maximize memory efficiency for double buffering (Fig. 7-28a). The dashed lines in the figure represent available space for double buffering. The 4x4 matrix blocks are placed into memory as shown.

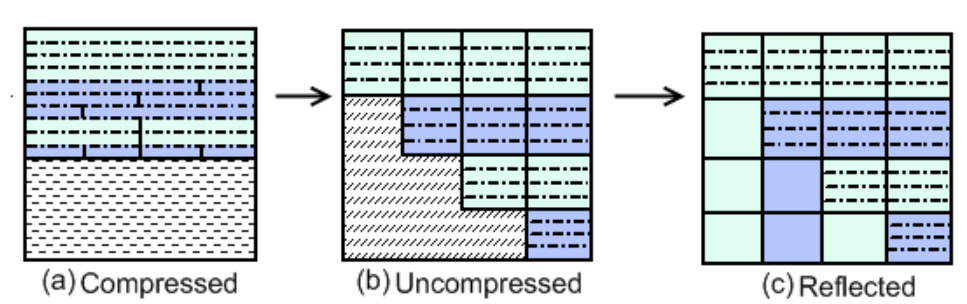


Figure 7-28: Kernel Generation on the SPEs

The algorithm for generating the compressed kernel is shown in Listing 32.

```
Given problem size, calculate size of compressed kernel

compressed_kernel       ←LS_Rsv
nonexpanded_rows        ←LS_Rsv(4 * dimension)
expanded_rows           ←LS_Rsv(4 * dimension)
DMA_list_buffer         ←LS_Rsv(problem size)
norms                   ←LS_Rsv(problem size)
vlxs                    ←LS_Rsv(problem size)
outputs                 ←LS_Rsv(problem size)
sparse_vec_buff         ←LS_Rsv( remaining space on LS)

Download DMA lists into DMA_list_buffer
Download norm and vlx data into norms and vlxs

Iterate over rows of blocks 4 at a time
        DMA.L in the next 4 sparse vectors into nonexpanded_rows
        Expand the sparse vectors into non-sparse form into expanded_rows
        Generate 4x4 block (don't need any more training vectors; this is a block on the diagonal)
        Iterate over the remaining sparse training vectors for this row (loop until all processed)
                DMA.L in as much sparse vectors as can fit into sparse_vec_buff
                Generate Kernel matrix elements using expanded_rows, sparse_vec_buff, and outputs
        Repeat
Repeat
```

Listing 32: Kernel Generation on the SPEs

Essentially, the outer loop moves the generation sequence down the matrix and the inner loop moves from left to right. The outer loop grabs four training vectors at a time and

decompresses them into nonsparse form. The inner loop does not decompress vectors, and thus brings in as many sparse vectors that can fit into the *sparse_vec_buf*. The number downloaded is calculated on the fly.

Two DMA lists are used for downloading sparse training vectors: one for the sparse values, and one for sparse indices. DMA lists are generated by the PPE since it is "aware" of the addresses for the start of all the sparse training vectors in the sKernel object.

Once created, the compressed matrix is uncompressed (Fig. 7-28b) and reflected (Fig. 7-28c), overwriting any temporary buffer space that was used during the element generation step. The quick LS accesses and large registers make this process relatively quick using proper vector shuffle operations. Note that there is a need for two additional rows in memory by the decompression algorithm. This is a small price to pay, however.

7.3.4 TrainingIndices Class

A problem can always be generated given a set of training indices. Given a global problem having uniquely indexed training vectors, the generation of subproblems can be performed by algorithmically distributing and combining these indices and using them to gather the global problem data. The SVM solvers in a Cascade SVM act as filters, with outputs being a subset of the input indices. It was concluded that a single job, or problem, should be represented as a set of indices to be processed. To provide a means of storing the result of a job, each of the indices in the set should hold some information about it, such as whether it represents a support vector or not. The storage of these indices should exhibit quick random access and iteration and be memory efficient. The index set should be capable of spawning subsets of itself, and remembering those indices which were spawned off. It should also be capable of being merged with other sets.

The *TrainingIndices* class was written to cover all these requirements and be flexible enough to allow for multiple jobs to train on the same training vectors simultaneously. When an instance of *TrainingIndices* is first created, it requires fast random access to the index set. However, when placed in the job queue, it needs to take up as little memory as possible. Two methods of storing data were devised: compressed (sparse) and uncompressed. A newly created instance uses the uncompressed format in which the data is represented as an internal byte array with a size of the global training set. In other words, there is a one-to-one relationship between all global training vectors and all bytes. Once the training indices are set, the dataset is compressed. In compressed form, two arrays are used. The first is similar to the uncompressed form but contains only indices for those vectors in this set. The second array holds the actual index values for each of the elements in the first array. Converting Compressing and uncompressing is done by calling the *CompressIdxs* and *UncompressIdxs* functions on the *TrainingIndices* instance.

No matter which storage mode, the information encoded in each byte element is shown in table 6.

Attribute	Type	Description
Original Set	*Boolean*	All indices that were part of the original input set have this attribute set.
Current Set	*Boolean*	A subset of the original set. The filtered support vectors have this attribute set.
Cass +	*Boolean*	If this training vector belongs in the positive set.*
Class -	*Boolean*	If this training vector belongs in the negative set.*
Reserve Count	*Count*	How many times this index has been reserved.

Table 6: Bit encoding of single one-byte element contained in TrainingIndices object

7.3.4.1 Distribution

Each node in the Cascade SVM tree holds a *TrainingIndices* instance, whether the problem is solved on that node or if it is farmed out to child nodes. The simpler case, in which the number of indices is small enough to generate a problem, the *TrainingIndices* instance is queued in the job queue. If, on the other hand, it is too large, a child Cascade421 instance is spawned from the node and the four nodes of the first layer of the child Cascade421 are initialized. Initializing these nodes is a matter of distributing the indices from the aforementioned *TrainingIndices* instance (now referred to as the parent *TI*).

First, a new instance of a *TrainingIndices* object is created at each of the four nodes on the first layer of the child *Cascade421*. Next the *ReserveSubsetOfRatio* function of the parent *TI* instance is called using the newly created *TIs* as parameters, thus copying the byte values into the new *TIs* and marking them reserved in the parent *TI*. Once a child *TI's* data is set, it is compressed. The result is four newly created sets of training vectors from the parent set. This distribution process is iterative in that if the subset sizes are still too large, they are further distributed into children Cascade421s.

7.3.4.2 Merging

Not only do training indices need to be distributed, but they need to be merged as well. In the situation that both dependencies for a node in a *Cascade421* instance are met, their resulting support vectors need to be merged to create the training indices set for that node. Performing this task is as simple as creating a new *TrainingIndices* instance for the node, and calling the *MergeWith* function on it passing each of the dependencies' *TIs* as parameters in succession. The *MergeWith* function only extracts those indices that are have the Current Set attribute set; those that turned out to be the support vectors. Once done, it is compressed as before.

Once again, when the indices are merged, they may form a new set that is too large for a single solver. In this case a child *Cascade421* instance is created for that node and the distribution algorithm is followed as described in the previous section.

7.3.4.3 Updating Parent TrainingIndices

For any node that was "farmed out" into a child *Cascade421* object, the *TrainingIndices* will eventually need to be updated with the results of that object. While iterating over nodes for a *Cascade421* instance, if a node was "farmed out" at some point and it is now finished, the updated *TrainingIndices* instance of that node is used to update the *TI* of this *Cascade421* object by calling the *UpdateFromSubset* function on the *TI*. Also at this point, the reservation counts for the *TI* associated with this *Cascade421* are updated from

the same subset by calling the *UnreserveFiltered* function. The implication of this is that any *TI* object associated with a *Cascade421* object has no reserved indices once all child *Cascade421* objects are finished.

7.3.4.4 Performance Optimizations

Upon running the Cascade Filtering algorithm, it became apparent that the task of processing dependencies, specifically managing the *TrainingIndices* objects, took a lot of time relative to the time it took to process the job queue. The issue was magnified when the dependency tree grew to large sizes. It would be very difficult to move the workload onto the SPEs. Instead, effort was placed into optimizing the *TrainingIndices* functions.

*uchar8_t *TrainingIndeces::GetIdxAttributes(uint32_t idx) const;*

The first function to be optimized was *GetIdxAttributes* due to the frequency of its execution. This function uses the *idx* as a key into the internal data, and returns the byte containing all the attributes for the corresponding training vector. The value 0 is returned if the vector is not in the set. Clearly, if the data is uncompressed, returning the value is as easy as accessing the idx^{th} value in the array. The optimization applies to the compressed case. Originally, finding the correct position in the compressed array involved sequentially comparing each element in the sparse index array to the *idx* parameter until one was found. Because the sparse index array is already sorted, a binary search was implemented.

uint32_t TrainingIndeces::CompressIdxs()

Next, the compression function *CompressIdxs* was optimized to utilize the Altivec instruction set. The original algorithm was simply to iterate over the uncompressed array looking for attribute bytes having the Original Set attribute set, and copying those values to a new sparse array. To improve the speed, the vector function *vec_any_ne(vector)* was used. This function takes a floating point vector as a parameter and returns a scalar signifying if any of the values are not zero (any attribute byte in the set cannot be non-zero and not have the Original Set attribute set). The benefit of this optimization was the iteration of the uncompressed kernel four elements at a time, as opposed to just one.

The third optimization affected several functions, but was designed to reduce the time for iterating over uncompressed data sets. The idea was to keep track of the very first index that contains a non-zero value. This entailed updating any function that modified the data set by adding or removing elements. By always knowing the first valid entry, any search through the array could start at that value instead of zero. This optimization was probably the most efficient due to the clumping tendency of the indices.

Finally, a similar optimization to the previous one was keeping track of the number of indices in the original set. Doing this allowed for early stopping of any iteration sequence over an uncompressed set rather than searching until the end of the array. Essentially, together with the last optimization, searches over uncompressed sets were bracketed.

The addition of these optimizations made the PPE's performance more closely matched with the performance of the SPEs.

Chapter 8: Test Methodology and Results

8.1 Chapter Introduction

This chapter describes the testing methodology and presents the results obtained from the final product, as well as several notable improvements during the work's progress. It is divided into two sections, one for the Multi Layer Perceptron, and one for Support Vector Machine implementations.

While the Cell processor is a hyped as a fast machine capable of many gigaflops per second, it is very sensitive to the implementation of the software and requires careful bottleneck analysis. A lot of work was put into analyzing the performance of individual sections within the implementations. The results that follow were achieved by a solid 10 months of development time.

8.1.1 Test Systems and Setup

In addition to the Playstation 3 system, which includes a PowerPC processor (the PPE), an Intel Pentium 4C desktop machine was used as a means to collect data for the single threaded implementations of the software. The Pentium 4C system is one of the last processors produced by Intel that consists of a single core and was therefore believed to be a fitting system for comparison. Both machines were loaded with the Fedora 9 Linux operating system. To maximize resources, no graphical user interface or windowing system was running while executing the tests. The specifications for both systems are shown in Table 7.

	Playstation 3		**Pentium 4 Desktop**
Processor	*PPE*	*SPEs*	32 bit Intel Pentium 4 HT 2.4C 800Mhz Bus (clocked to 3.0Ghz) 12 KB Decoded Microinstruction Execution Trace Cache 8 KB DCache 512 KB L2 Cache
	64 bit PowerPC 3.2Ghz 32 KB L1 ICache 32 KB L1 DCache 512 KB L2 Unified Cache	128 bit 3.2Ghz 256 KB Software Managed Local Store	
Memory	256 MB		1.5 GB
Hard Drive	SATA2		PATA

Table 7: Test Hardware

When compiling the source code for the Cell, the *ffast-math* compiler option was used to force SPE floating point optimizations, although it should not have made much of a difference since using the SPU intrinsics to choose SIMD instructions forces the use of optimized operations. In addition, the latest Cell-optimized Mathematical Acceleration SubSystem library (MASS) was used on the SPEs [65]. This library provides accelerated floating point functionality at the expense of accuracy [66]. A speedup of the optimized functions was analyzed in [67]. On both systems, an optimization level of 3 (*-O3*) was used.

8.1.2 Timing Methodology and Accuracy

The Cell Simulator was run on the Pentium machine running the officially supported Fedora 9 Linux distribution. The speed at which the simulated system ran the compiled binaries was at least two orders of magnitude slower than when run directly on the Cell hardware. To speed up the optimization process, a quick method for finding sections worthy of optimization was needed (preferably done on the hardware). Due to the lack of support for reading of the low level registers for performance monitoring of the PPE on the Playstation 3, the GNU *gettimeofday()* function was found to be a good alternative. This function is known to be accurate in the microsecond range. A simple profiling utility module was written utilizing this function that allowed for the measuring of total execution time and total hits for multiple portions of the code. At the end of execution, a table was printed displaying the statistics for the profiled sections. The SPEs, on the other hand, do allow access to a special register known as the *decrementer*. The decrementer is a downward counter that ticks at the system's timebase frequency (79.8 MHz on the Playstation 3). Similarly to the PPE, wrapper macros were written to support multiple profiling sections. The profiling code was used to narrow down on code sections that were taking up the most execution time. Once found, these sections were profiled in the simulator so that the causes of the slow downs and potential for optimization would be learned.

```
#define RESET_DEC()     spu_write_decrementer(0xFFFFFFFF)
#define GET_DECTICKS()  (0xFFFFFFFF - spu_read_decrementer())
```

Listing 33: SPE Decrementer register access macros

```
#define CLK_MARK_START(a)     {gettimeofday(&tv,NULL);
                               t0[(a)] = tv.tv_sec * 1000 + tv.tv_msec;}
#define CLK_MARK_END(a)       {gettimeofday(&tv,NULL);
                               t1[(a)] = tv.tv_sec * 1000 + tv.tv_msec;}
#define CLK_INTERVAL(a)       (t1[(a)] - t0[(a)])
#define CLK_PRINT(a)          printf( " %u ms\n", CLK_INTERVAL(a) )
```

Listing 34: PPE profiling macros

Anywhere that the performance did not meet expectations, the reasons for the results are hypothesized and supported with either additional analysis, or external research.

8.2 Multi Layer Perceptron

8.2.1 Dataset

The implementation of the Multi Layer Perceptron for this work was tailored for a specific data set, namely the MNIST database of handwritten digits. This dataset is freely available on the Internet [68] and is a subset of a larger set available from NIST (National Institute of Standards). The MNIST database contains 60,000 training samples and 10,000 test samples.

The MNIST dataset input set consists of 28x28 pixel images which were generated by normalizing the NIST versions into a 20x20 pixel box while preserving their aspect ratio, and centering them in the 28x28 canvas by their center of mass. While the original images consisted of purely black and white pixels, the MNIST dataset contains grey levels as well due to the anti-aliasing effects of the resizing operation. For a full description of the set, as well as some previously documented results, see [68].

For the implementation of this thesis, further dataset processing was necessary due to complications arising from padding of data for use by the SPEs. The original 28x28 images were resized to 25x25. The problem existed in the connection between the last convolution layer and the first fully-connected layer. Due the fact that the pixels in the output feature map were treated as contiguous neuron output values in memory, the output feature maps needed a width of a multiple of four for no padding to occur. With subsampling enabled and kernel sizes of 5x5, the feature map sizes of the second convolution layer were 11x11, and the feature maps on the third and final convolution layer were 8x8, which did not require padding.

8.2.2 Network Structure

While the implementation was designed to train on the MNIST database, it is flexible enough to support any combination of input, hidden, and output layers. Convolution layers are optional, but must appear before fully-connected layers. Attributes such as the size of the kernels, number of neurons in the fully-connected layers, weight randomization flag, and method of learning (incremental or RPROP) are selectable, but currently require the application to be recompiled. In this thesis, the following network was used. It was chosen based on an existing implementation at [69].

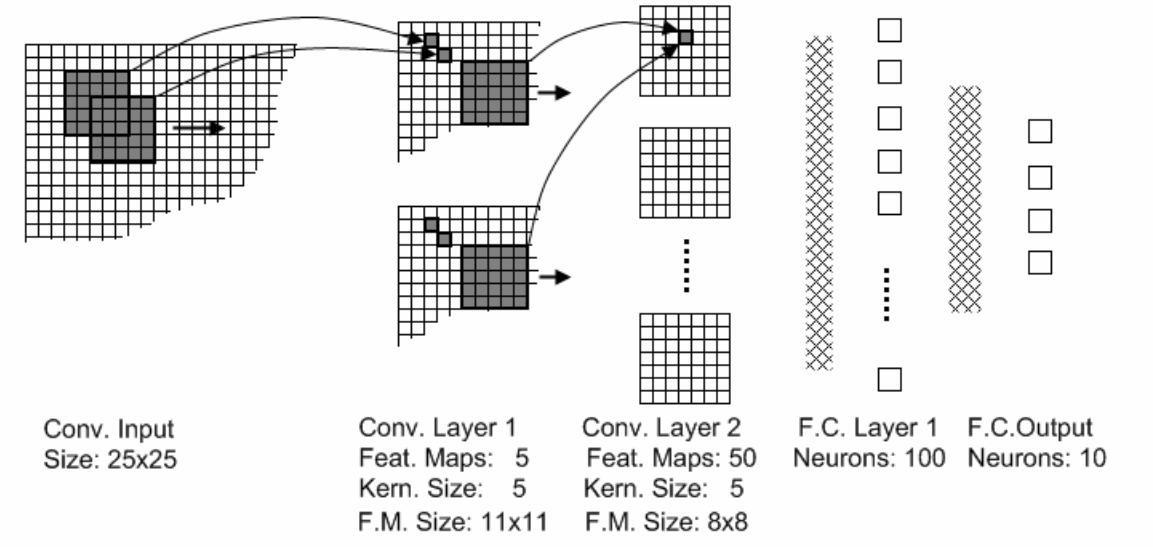


Figure 8-1: MNIST network structure

8.2.3 Testing Strategy

While there exist at implementations of the MLP that support convolution layers [69], all code was written from scratch. Two versions of the implementation were written – one single threaded (*nn_single*), and one targeted for the Cell Processor (*nn_cell*). Writing and debugging two versions side by side reduced the probability of implementation error, and allowed for the verification of correctness by comparing data within common sections in the code. The simpler single-threaded version required no special data representation. This version was run on the PowerPC based PPE on the Playstation 3 as well as the Pentium 4C processor. The next version was designed to take specific advantage of the Cell processor by following many of the design strategies explained in Chapter 6. Both versions were implemented and compiled into a single application so that the two could be used interchangeably at different sections of the algorithm (although doing so required conversion of data between the different data representations). The extra overhead, however, was not an issue during debugging. All Cell-specific source code was disabled when compiling on the Pentium system.

When collecting execution data for comparison between the two systems, to increase fairness initial layer weights were loaded from a common file to have a common starting point. Special care had to be taken to convert between little endian and big endian floating point representations. In both versions single precision floating point datatypes were used.

8.2.4 RPROP Learning Results

The first experiment consisted of four executions of the algorithm, differing only by the compiler and hardware. The full-sized 60,000 element training set was used with RPROP chosen for the learning method. The RPROP settings were as follows (see section 2.6.6.1 for explanations):

RPROP_INCREASE_FACTOR	1.1
RPROP_DECREASE_FACTOR	0.6
RPROP_UPDATE_VALUE_MAX	3.0
RPROP_UPDATE_VALUE_MIN	1e-8
RPROP_INIT_UPDATE_FACTOR	1e-3

Table 8: RPROP settings A

	Compiler	**Learning Mode**	**Iterations**	**Time**	**Time/Iteration**	**Test Set Accuracy**
Cell final	XLC	RPROP	264	2h47m	38.18s	96.72
Cell final	GCC	RPROP	268	2h54m	38.96s	96.88
Cell final (fdrpro)	GCC	RPROP	268	2h54m	38.96s	96.88
Pentium4	GCC	RPROP	296	13h6m 14h17m	2m39s (159s) 2m53s (173s)	97.56

Table 9: Results for RPROP settings A

The speedup per iteration of the Cell implementation is 4.55. The single threaded implementation did produce a slightly higher accuracy on the test set. It is difficult to say whether that is due to the non-IEEE compliant floating point representation on the SPEs, or that there is an actual inconsistency in the calculations on the Cell implementation due to some an error in the code. The difference, however, is small enough to be ignored. The XLC compiler made little difference in the final execution time, but resulted in a lower test set accuracy. Performing dynamic analysis using the *fdrpro* tool resulted in no speed up at all. The speedup, compared to the Pentium 4, is about 4.17, which is lower than expected. To reinforce the identical functionality of the two implementations, the mean squared error at the output was recorded over the entire execution for both implementations. The two convergence plots are shown overlaid in Fig. 8-2.

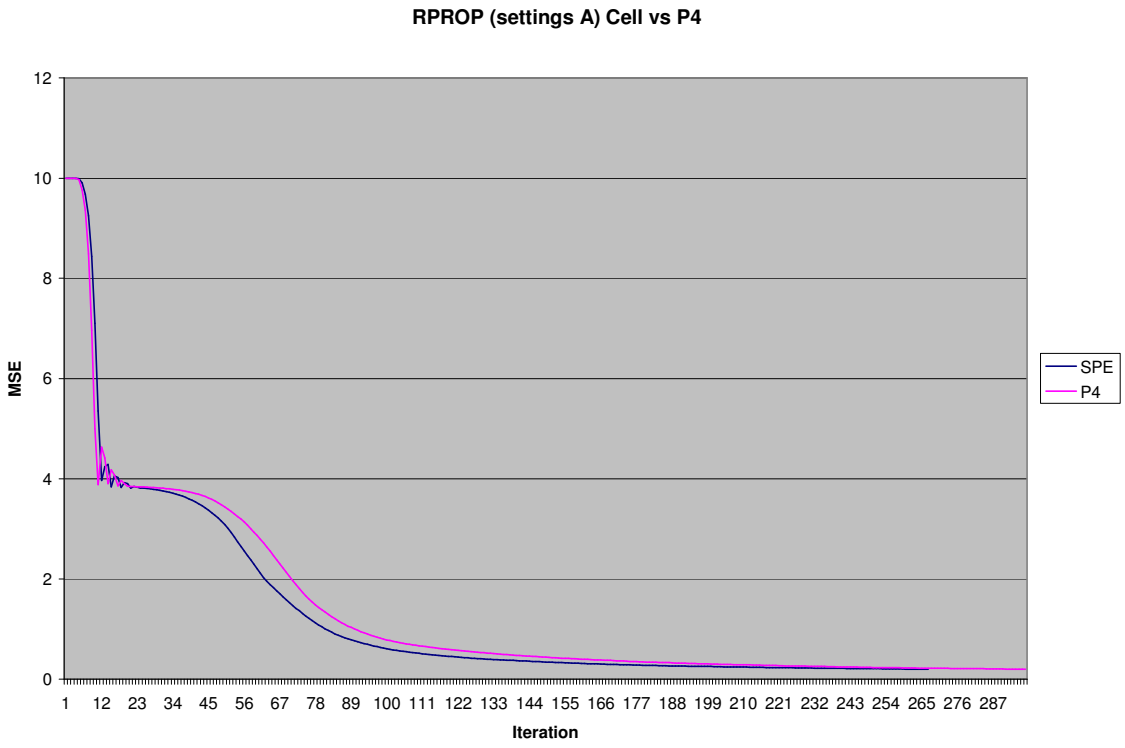


Figure 8-2: Comparison of convergence rates between nn_single and nn_cell

The slight variation in the convergence iterations between the two architectures is acceptable due to the nature of the Backpropagation algorithm. A possible explanation for the quicker SPE convergence is that it did not gain as much momentum in the early steep descent, leading to a smaller overshoot, and quicker recovery at around the 12th iteration.

The sudden step at around the 12th iteration is likely due to an overshoot by RPROP Backpropagation method. The slow recovery that follows is a feature of this algorithm. Effort was placed to eliminate this staircase effect by tweaking the RPROP parameters, but no solution was found. Here is another experiment using the recommended RPROP settings [15].

RPROP_INCREASE_FACTOR	1.2
RPROP_DECREASE_FACTOR	0.5
RPROP_UPDATE_VALUE_MAX	3.0
RPROP_UPDATE_VALUE_MIN	1e-8
RPROP_INIT_UPDATE_FACTOR	1e-3

Table 10: RPROP settings B

	Learning Mode	Iterations	Time	Time/Iteration	TestSet Accuracy
Cell final	RPROP	351	3h54m	40.00s	96.70
Pentium4	RPROP	424	21h10m	2m59.72s	97.40

Table 11: Results for RPROP settings B

Fig. 8-3 shows the convergence graph for the Cell version of the code.

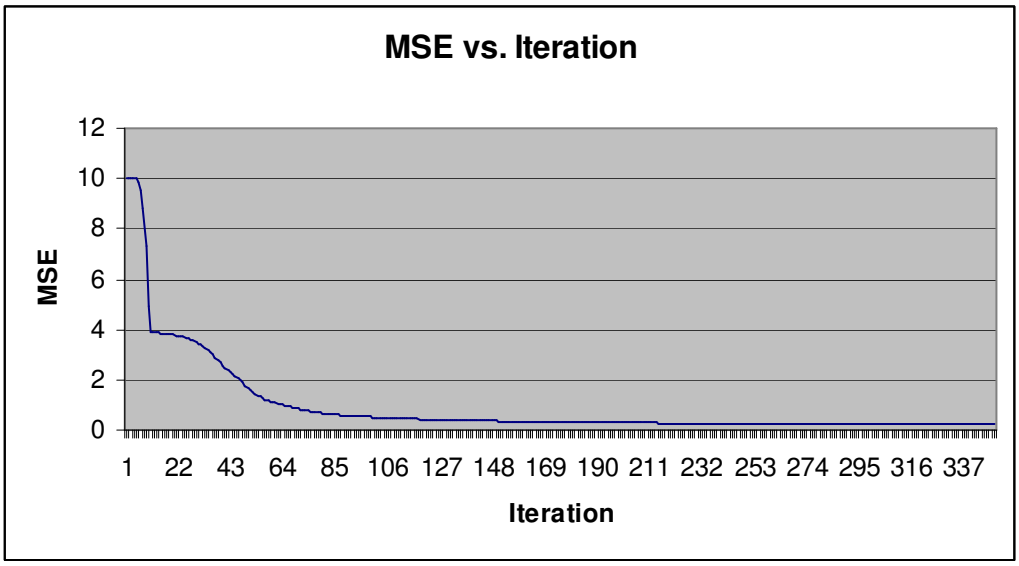


Figure 8-3: nn_cell convergence graph for RPROP settings B

As a final RPROP experiment, the third execution used very aggressive RPROP parameters.

RPROP_INCREASE_FACTOR	1.2
RPROP_DECREASE_FACTOR	0.5
RPROP_UPDATE_VALUE_MAX	50.0
RPROP_UPDATE_VALUE_MIN	1e-8
RPROP_INIT_UPDATE_FACTOR	1e-3

Table 12: RPROP settings C

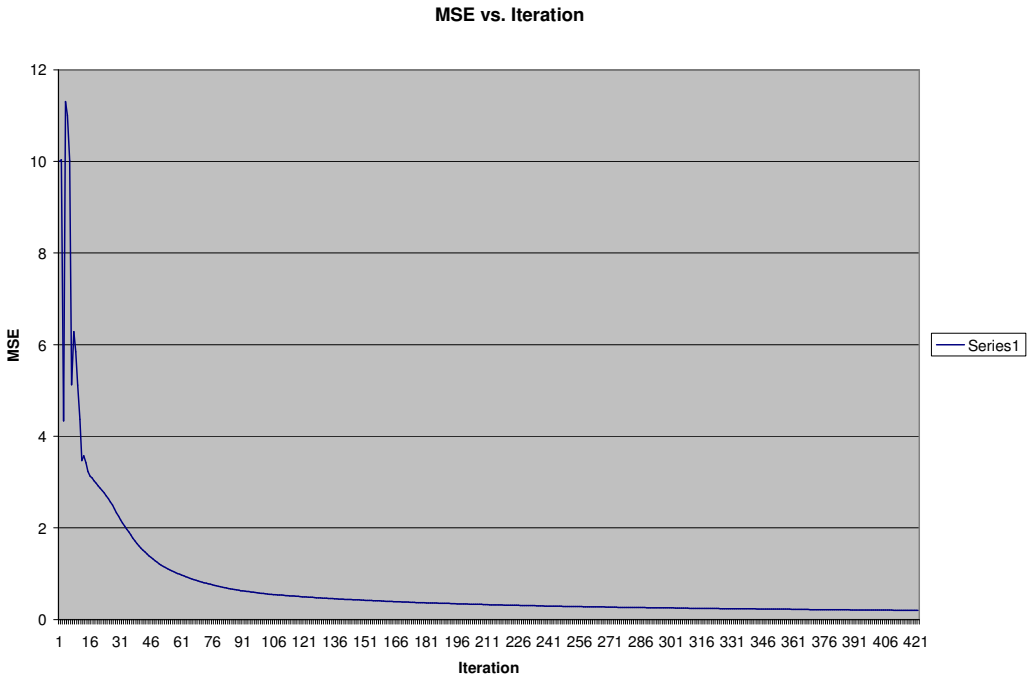


Figure 8-4: nn_cell convergence graph for RPROP settings C

The staircase effect is gone in this example. There are some oscillations at the start which quickly stabilize. Nevertheless, the conversion took 422 iterations, which is quite an increase from the two previous learning experiments.

8.2.5 Incremental Learning Results

Next, a single test was run using incremental, rather than batch (RPROP) learning. As discussed in Chapter 2, in this mode, the weights are updated per training sample, rather than per training epoch. Results are shown in table 13.

	Compiler	Iterations	Time	Time/Iteration	TestSet Accuracy (%)
Cell final	GCC	5	3m35s	43s	97.81
Pentium4	GCC	5	19m31s	3m54s	98.03

Table 13: Incremental learning results

The speedup of nn_cell over nn_single was 5.45. This is very similar to the speedup observed when training using RPROP. The number of iterations is significantly smaller, however. See the next section for possible explanations for this result.

8.2.6 Discussion

The speedup of the Cell implementation of the MLP algorithm was notable but not as great as expected. There is a possible explanation, and may have to do with the top-level Cell implementation design. To recap, each SPE runs a kernel that listens for incoming commands and executes them when they arrive. The first test performed in search of the performance bottleneck was the profiling of the time spent waiting for the command and

the time spent serving it. The two times were compared and found to be quite unbalanced. It was discovered that each SPE spends much more time just waiting for something to do rather than actually doing work. As a means of collecting wait and execution times, the SPE decrementer macros were used. One decrementer tick is approximately equal to 12.5 nS. Table 14 shows the average wait vs. service times for each of the important commands and each of the layers.

	Forward Propagation Time (uS)			**Backpropagation Time (uS)**		
Layer\	Wait	Exec	W/E	Wait	Exec	W/E
L1 (conv.)	163.70	4.94	33.15	66.83	8.94	7.48
L2 (conv.)	135.61	69.85	1.94	83.62	43.35	1.93
L3 (full)	49.13	30.64	1.60	78.71	60.13	1.31
L4 (full)	71.6	2.08	34.51	130.14	12.61	10.32

Table 14: Average wait and service times for SPE-offloaded functions

Another problem with the implementation is the method for division of work among the SPEs given the network shown in Fig. 8-1. The initialization step divides the workload within each layer. The actual division of work for all the layers for the MNIST problem is as follows:

	L1 conv. (OFM's)	**L2 conv. (OFM's)**	**L3 full (fw/bp jobs)**	**L4 full (fw/bp jobs)**
SPE0	1	9	2/2	1/1
SPE1	1	9	2/2	0/0
SPE2	1	9	2/2	0/0
SPE3	1	9	1/1	0/0
SPE4	1	9	0/0	0/0
SPE5	0	5	0/0	0/0

Table 15: Per-layer workload distribution

There are several steps at which the SPEs are underutilized. For example, the output layer is only processed by a single SPE while the others five sit idle. Not only that but the task itself is executed almost 35 times faster than the time waiting for the command (not including the input and output DMA transfers). The processing of this layer is therefore highly inefficient. The first fully-connected layer is also not well distributed. Two SPEs sit idle, and one does half the work of the other three active SPEs.

To further examine the workload imbalance, the application was run on the simulator and analyzed using the SPE Visualization charts. This visualization tool provides a graphic summary of what each SPE is doing over time. Data displayed includes channel communication, DMA transfers, and execution of code. Unfortunately, the code execution graphs are inaccurate due to the method by which the tool collects data (the instruction type at predefined intervals is detected and accumulated). However, it did show that the SPEs spend much time stalled on channel read instructions rather than performing useful work.

The number of iterations needed to converge using incremental learning has dropped significantly compared to that using the RPROP method. This may seem inaccurate at first, given the praise that batch learning methods receive in literature. There are a couple of possible explanations, however, for the big performance difference. When doing incremental learning, it is recommended to randomize the application of training examples to the input of the network. This is something that was not exercised in this work. Not doing so gives a higher preference to the first examples in the set which has the effect of decreasing the training set size. Incremental learning updates the weights 60,000 per epoch. Batch learning updates the weights once per epoch. This implies that the RPROP learning converged after about 260 weight changes. The incremental learning converged after 78,000,000 weight changes. The fact that there is some similarity of the training inputs, given that they were centered by their center of mass implies that the weight changes would follow a similar path for each one. This would naturally lead to quicker convergence.

8.3 Gradient Projection-Based Decomposition Technique

The GPDT algorithm source code was obtained from [70]. For all single threaded tests the original GPDT algorithm was used after being modified to replace all double precision floating point operations into their single precision versions. Early experimentation showed that doing so did not hamper the accuracy performance of the Dai-Fletcher Projected Gradient method QP solver. The Generalized Variable Projection method which was also available in the GPDT package, however, suffered from convergence issued after making this change and was therefore not implemented on the Cell.

8.3.1 Application Usage

The GPDT application takes several parameters on execution. They are mostly carried over from the original GPDT source code. Table 16 lists them all.

Parameter	Type	Description
-g	*float*	parameter *g* in Radial Basis Function kernel
-c	*float*	parameter C for SVM classification; trade off between training error and margin
-q	*int*	size of the QP-subproblems per iterations
-n	*int*	maximum number of new indices entering the working set in each iteration
-m	*int*	cache size in MB

Table 16: cell_gpdt command line parameters

In the experiments below, unless otherwise noted, the size of the QP subproblems is set to 400 (the maximum supported) and the parameter *n* is at 200. The cache is generally turned off except in the MNIST set whose smaller size makes it beneficial to leave it on.

8.3.2 Data Set Selection

The choice of the datasets was guided by the developers of GPDT themselves, who used them in their own work for data collection [35]. The Gaussian kernel was used as well.

The GPDT application is based on the SVM^{light} [40] application, and uses special data file formats. Unfortunately, the authors did not include their larger test sets with the GPDT package. As a result the datasets were obtained from the UCI online database at [72] and needed to be converted into the SMVL format manually. This was done by writing special dataset conversion tools.

8.3.2.1 MNIST Handwritten Digits Dataset Results

The first experiments were run on a smaller data set that was bundled with the original PGPDT source code package. The dataset is derived from the same set as that used by the MLP implementation in this work. Unlike the MLP algorithm, however, the SVM (due to its binary classification nature) is trained to recognize only the digit 8 (that is, whether the input is an 8 or it is not an 8). This simple test set was used extensively throughout development for validation purposes.

Training Set Size	10,000
Input Dimension	784 (all continous)
Sparsity	20%

Table 17: MNIST dataset

Table 18 displays results collected in [35] (italicized) as well as those collected in this work. Those results obtained from literature utilize double precision operations. Those obtained in this work are based on single precision operations. Note that in the calculation of time, actual data loading/reformatting is not included as it takes a significant portion of the total execution time (appx. 7 seconds).

	obj	***b***	***SV***	***BSV***	***iter***	***time***	***spd up***
Intel Itanium 2 @ 1.3Ghz	*-2083.576*	*6.4401*	*1590*	*39*	*55*	*20.6s*	*ref.*
Xeon Cluster (8 proc)	*-2083.576*	*6.4401*	*1590*	*39*	*54*	*5.6s*	*3.68*
Pentium 4	-2083.576	6.4401	1591	39	46	183.9s	0.11
Pentium 4 (128MB Cache)	-2083.576	6.4401	1591	39	46	28.88s	0.71
PS3 PowerPC	-2083.576	6.4401	1591	39	53	495.87s	0.04
PS3 PowerPC (128MB Cache)	-2083.576	6.4401	1591	39	53	78.5s	0.26
Cell	-2083.581	6.4393	1591	39	48	15.3s	1.33
Cell (128MB Cache)	-2083.581	6.4393	1591	39	48	2.01s	10.25

Table 18: GPDT results for MNIST dataset

8.3.2.2 Forest Covertype Dataset Results

The forest covertype dataset [71][72] can be used to train a classifier to predict forest cover type from cartographic variables only (no remote sensed data). This is useful for natural resource managers responsible for developing ecosystem management strategies who may not have access to detailed information. The task was to train the classifier to distinguish class 2 (Lodgepole Pine) from the other classes.

Total Samples	581012
Input Dimension	54
Sparsity	78%

Table 19: Covertype dataset

The GPDT input parameters were:

-g	2e-5
-c	10

In the first experiment, the classifier was trained on a reduced 100,000 element training set by random selection of the entire set. The file reading and writing time was included in the results (Table 20).

	PC	**Cell**
Time	174m24.968	3m52.793
Iterations	732	746
Objective Func. Val	-51069.564517	-51096.012586
Threshold b	0.097316075	0.09706863
SV	12892	12898
BSV	5182	5183

Table 20: Reduced covtype training results

Next, the entire set of 581012 elements was used as the training set. The results are shown in Table 21.

	PC	**Cell**
Time	7928m37.321s (5 days, 5.6h)	234m55.683s (3.9h)
Iterations	7272	7405
Time/Iter.	1m5.418s	1.904s
Objective Func. Val	-520354.760315	-520757.880704
Threshold b	0.23636081	0.23667677
SV	79793	79852
BSV	59687	59717
Accuracy on Test Set	97.90%	97.85%

Table 21: Full covtype training results

The test set that was used to collect accuracy information was generated by randomly sampling 2000 elements from the training set.

8.3.3 Cascade SVM Results

For the following results, the full covtype dataset was used. The test set was generated by randomly selecting 100,000 samples from the full set. The controlled parameter was the desired filter ratio, as introduced in section 7.3.2. Because of the large amount of data collected, two tables were generated. Table 22 is more algorithm-oriented, providing the total training time (for the combined filtering and GPDT steps), the number of filtering iterations (number of times the dependency tree was processed), number of QP problems solved, number of support vectors at the output of the Cascade SVM filter, and finally the number of iterations performed by the GPDT algorithm.

Filter parm.	total t.	Filter Iter.	Num Solver.	SV (Filtered)	Filter Eff.	GPDT Iter.
40%	33m45.410s	7	15933	158220	0.727682	3561
45%	33m40.848s	7	16047	158196	0.727723	3566
50%	33m6.522s	8	16233	157792	0.728419	3475
55%	33m8.820s	8	16325	157604	0.728742	3498
60%	32m28.844s	11	16378	156540	0.730574	3449
65%	31m16.914s	14	16302	152228	0.737995	3416
70%	31m20.051s	16	16302	152348	0.737789	3414
75%	31m3.253s	15	16340	152116	0.738188	3375
80%	30m10.072s	13	16461	151508	0.739234	3302
85%	30m7.299s	16	16731	150132	0.741603	3335
90%	27m6.120s	29	19703	141700	0.756115	3113
92%	24m16.098s	37	23129	129412	0.777264	2960
95%	22m8.724s	50	29643	122264	0.789567	2797
98%	13m33.637s	138	66667	79556	0.863073	1946

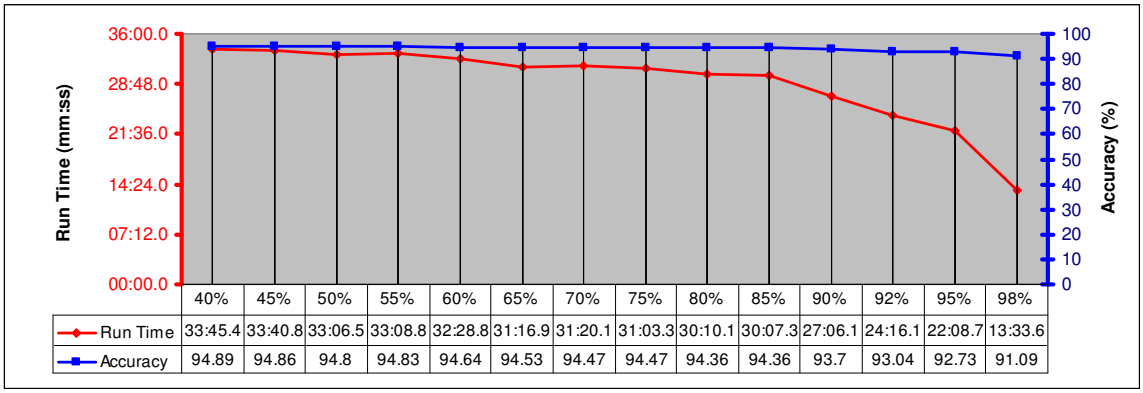
Table 22: Cascade SVM+GPDT Algorithm-oriented results

The next table (Table 23) is more problem specific providing information about the final objective function value, the threshold value, the number of support vectors found, the number of bounded support vectors found, and the accuracy of the trained system on the test set.

Filter parm.	Object. Func.	Thresh. b	SV	BSV	Accu
40%	-255523.818208	0.18822893	43094	25632	94.89
45%	-255951.859882	0.18796756	43128	25615	94.86
50%	-254367.270926	0.18956316	42933	25437	94.80
55%	-254156.867370	0.18622063	42916	25469	94.83
60%	-253161.731203	0.18778661	42647	25348	94.64
65%	-247177.205784	0.17146861	41807	24653	94.53
70%	-247090.356186	0.16990008	41757	24677	94.47
75%	-246088.874525	0.17738432	41681	24524	94.47
80%	-244398.347347	0.17420151	41464	24447	94.36
85%	-240862.417043	0.17336849	41063	24012	94.36
90%	-222922.765339	0.16443646	38819	21951	93.70
92%	-204001.430865	0.15644024	36358	19833	93.04
95%	-192689.710254	0.15235362	34876	18675	92.73
98%	-131428.309363	0.11606202	26560	11778	91.09

Table 23: Cascade SVM+GPDT Problem-oriented results

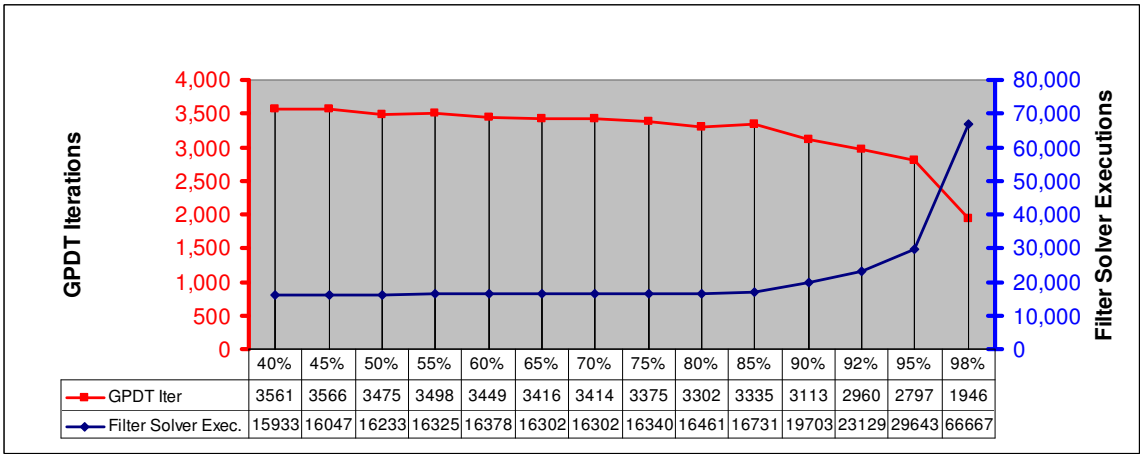
Fig. 8-5 graphically shows the effect that the desired filter ratio had on the total run time and final accuracy on the test set.



	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	92%	95%	98%
Run Time	33:45.4	33:40.8	33:06.5	33:08.8	32:28.8	31:16.9	31:20.1	31:03.3	30:10.1	30:07.3	27:06.1	24:16.1	22:08.7	13:33.6
Accuracy	94.89	94.86	94.8	94.83	94.64	94.53	94.47	94.47	94.36	94.36	93.7	93.04	92.73	91.09

Figure 8-5: Desired Filter Ratio vs. Total Runtime and Final Accuracy on the Testset

Fig. 8-6 shows the effect of the desired filter ratio of the number of GPDT iterations and the number of filtering QP solvers executed.



	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	92%	95%	98%
GPDT Iter	3561	3566	3475	3498	3449	3416	3414	3375	3302	3335	3113	2960	2797	1946
Filter Solver Exec.	15933	16047	16233	16325	16378	16302	16302	16340	16461	16731	19703	23129	29643	66667

Figure 8-6: Desired filter ratio vs. GPDT iterations and Filter Solver Executions

8.3.4 Discussion

The MNIST speedup was quite impressive, and showed that enabling the cache strategy did provide some speedup, as was expected for the smaller dataset. The capability of the PPE in relation the Pentium 4 is exposed as well, showing the poor capability when the SPEs are not utilized. To be fair, the original single threaded GPDT algorithm did not take advantage of the PPE's Altivec capability.

The speedup of the Cell implementation of the GPDT solver for the partial covtype dataset was an impressive 44.91. A smaller speedup of 33.75 was obtained with the full covtype dataset. One possible reason for the reduced speedup ratio with the larger data set is the increased involvement of the PPE in the iteration cycle. The PPE is tasked with the index replacement strategy, as well as the generation of several arrays during sub-problem generation. The large array memory accesses may be paying a toll on the unified L2 Cache of the PPE which is 512KB.

The results of the Cascade SVM show that the filtering step can have a very positive impact on the total training time and yet still maintain a high degree of accuracy on the test set. As expected, increasing the desired filter ratio placed more workload on the Cascade SVM portion and left less work for the GPDT portion. Unfortunately, because the Cascade SVM method is not perfect, the accuracy deteriorates the more vectors are filtered out. The time saved, however, makes up for it.

8.4 Other Results

8.4.1 Solver V1 vs. Solver V2

The second solver, as discussed in Chapter 7, was developed to take advantage of multiple SPEs and to allow for larger sub-problem sizes. The two solvers were written as modules so that toggling between the two would be relatively easy. For this test, the same exact problem was run on both solvers. The problem size of the V2 solver was lowered to that of the maximum of the V1 solver (192 indices). With this setup, the V2 solver perfectly balanced the workload among the SPEs, with each SPE being tasked with a total of 13 block jobs to generate, and storing 32 rows of the kernel per SPE. The size of each block was 16x16.

	Solver V1	**Solver V2**
Avg. Solver Time (mS)	366.3	1255.1
Avg. Num Iter	83.4	59.8
Total Iter	124	122
Total Time (S)	20.13s	18.00s

Table 24: Solver V1 vs Solver V2

The total execution time includes the data initialization which takes around 7 seconds. The effective speedup ratio of the solver is 3.40 which is a very favorable result.

8.4.2 Feedback Directed Program Restructuring

As mentioned in Chapter 5, the SDK includes dynamic optimization tool known as FDPR Pro that requires a “typical” workload so that the binary can be optimized. In order to obtain this “typical” workload, for all the following cases which have large training sets, a subset of the training set was used. Table 25 summarizes the results obtained before and after applying the FDPR Pro tool. The file loading time is included in all experiments.

Experiment	**Before**	**After**
GPDT MNIST dataset	10.467s	10.215s

Table 25: Feedback Directed Program Restructuring results

Unfortunately, the optimization could not be performed on the GPDT implementation due to its maximal use of the LS space. The FDPR Pro optimization sequence requires a portion of memory within the LS.

Chapter 9: Conclusion and Future Work

9.1 Chapter Introduction

This chapter is intended to provide the final word on the applicability of the Cell Processor for classification algorithms based on the experience gained in this work and on the final results. It also mentions any potential future work based on the lessons learned.

9.2 Multilayer Perceptron

The results obtained for the multilayer Perceptron implementation were below initial expectations especially due to the fact that matrix-vector multiplication, the heart of the MLP algorithm, has been shown to be very quick on the Cell. It is important to note that this was one of the first programs written on the Cell. The learning curve for programming performance applications on this processor is quite steep (see Chapter 5) and therefore this initial project was in part a training exercise. One rather advanced method that was not implemented anywhere in this code was multi-buffering. The absence of this technique is likely the source of the longer than expected execution time. As pointed out in the previous chapter, the job distribution did not expand well onto the 6 available SPEs. Simulator experiments also showed that the Local Stores were not being used to their full capacity.

Taking all these thoughts into consideration, and considering the fact that a speedup of 4-5x was still observed, the applicability of the Cell processor for the Multilayer Perceptron algorithm cannot be discounted. In fact, if incremental learning would not be required, it is hypothesized that parallelizing the learning task over an epoch, rather then over each layer, would produce much better performance. In batch learning, the weights are not modified until the end of an epoch. The parallelization strategy would simply distribute the training set among the SPEs and have each one run until the end with absolutely no communication necessary until the very end. In fact, the PPE could likely be left out of the process as well, making the algorithm perfectly suited for the high-bandwidth data throughput of the SPEs.

Having become more proficient at programming the Cell and learning from the mistakes made, it is safe to say that such an implementation of the MLP algorithm would be much quicker.

9.3 Gradient Projection-based Decomposition Technique

The Gradient Projection-based Decomposition technique is not as well structured as the MLP algorithm and was consequently more difficult to implement. Having some experience at developing software on the Cell processor, it took less time finding those sections which would benefit most from being optimized for the SPEs.

The final implementation of the algorithm was very well suited for the Cell for several reasons. During the problem generation and solving step, the kernel matrix (which at that step is the largest piece of data), was generated entirely by the SPEs and never left their Local Stores. This type of "disposable" data suits the SPEs very well as they do not need to perform additional DMA transfers. The inter-SPE data transfers made excellent use of the high bandwidth provided by the on-chip Elementary Interconnect Bus.

The other highly computational portion of the GPDT algorithm was the gradient updating step. This portion, similar to that of the generation step, also took advantage of generating "disposable" data in the form of the matrix kernel blocks. The incorporation of double buffering had the effect of keeping the SPEs busy. A big speedup was obtained by carefully optimizing the actual kernel element generation module. Of all the three algorithms implemented, this one gave best speedup.

Having placed the most effort into this algorithm, all original optimization ideas have been implemented to the fullest. The package, however, is only capable of performing the Gaussian kernel. As a future work, additional kernels might be implemented, allowing for the training on additional datasets.

9.4 Cascade SVM

The Cascade SVM algorithm looked very attractive from the start due to its asynchronous nature. Unfortunately, it is difficult to compare the Cascade SVM algorithm to those results obtained in literature due to the number of variables involved in the setup. Nevertheless, the results shown in Chapter 8 are extremely attractive, severely cutting down the total training time at the expense of accuracy. The two main difficulties with this implementation were designing the PPE-based dependency tracking code such that it could keep up with the SPEs and getting around the problem size limitation. A current limitation of the algorithm is its synchronous nature between the dependency tree scanning and problem queue processing. While the results have shown the potential of the Cell processor at Cascade SVMs, they are meant to be taken as a proof of concept. The SPE code for the solving of a size-limited QP problem has been developed. As a future work, additional connection schemes and dependency tracking methods should be examined.

9.5 Overall Conclusion

The work done in this thesis has shown great potential of the Cell Processor in the scope of classification algorithms. The greatest obstacle to developing efficient code for the Cell is the learning curve involved. The development time for the three algorithms of this work has taken nearly 10 months. The Cell processor can deliver some extremely attractive performance per cost ratios so long as the developer takes the time to become

familiar with the hardware and its methods. Recently, progress was made to open up Graphics Processing Units (GPUs) for general purpose programming. The methods involved are arguably still more complex than those for developing software on the Cell. In fact, the Cell processor fills the gap between the two extremes of the General Purpose Processor (GPP) and the GPU and similar application specific devices. It is almost guaranteed to provide performance closer to that of Application Specific hardware while requiring less effort, and likely at a smaller cost. The announcement of newer iterations of the Cell processor which will feature fully pipelined IEEE 754 compliant double precision floating point operations makes the Cell even more attractive for scientific applications.

References

[1] Insomniac Games, "More on SPU Shaders." [Online] 24 August 2008. http://www.insomniacgames.com/tech/articles/0108/more_on_spu_shaders.php

[2] W. S. McCulloch and W. Pitts, *A logical calculus of the ideas immanent in nervous activity*, pp. 15–27. Cambridge, MA, USA: MIT Press, 1988.

[3] F. Rosenblatt, *Principles of neurodynamics. Perceptron and the Theory of Brain Mechanisms.* Spartan Books. Washington D.C, 1962.

[4] M. L. Minsky and S. A. Papert, *Perceptrons: An Introduction To Computational Geometry*. MIT Press, Cambridge, Mass., 1969.

[5] P. J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences*. PhD thesis, Harvard University, Cambridge, Mass., November 1974. Reprinted in Werbos1994TRo.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," *Parallel distributed processing: explorations in the microstructure of cognition*, vol. 1, pp. 318–362, 1986.

[7] D. P. Bertsekas, *Nonlinear Programming 2nd ed.* Athena Scientific, 1995.

[8] B. T. Polyak, "Some methods of speeding up the convergence of iterative methods.," *USSR Comput. Math. and Math. Phys.*, vol. 4, pp. 791–803, 1964.

[9] B. T. Polyak, *Introduction to Optimization*. Optimization Software, 1987.

[10] J. Moody and C. J. Darken, "Fast learning in networks of locally-tuned processing units," *Neural Comput.*, vol. 1, no. 2, pp. 281–294, 1989.

[11] S. Saarinen, R. Bramley, and G. Cybenko, "Ill-conditioning in neural network training problems," *SIAM J. Sci. Comput.*, vol. 14, no. 3, pp. 693–714, 1993.

[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[13] K. Hornik, "Some new results on neural network approximation," *Neural Networks*, vol. 6, no. 9, pp. 1069–1072, 1993.

[14] W. S. Sarle, "comp.ai.neural-nets faq, part 3 of 7: Generalization." comp.ai.neural-nets, May 2001.

[15] M. Riedmiller and H. Braun, "Rprop – description and implementation details," tech. rep., University of Karlsruhe, 1994.

[16] S. E. Fahlman, "Faster-learning variations on back-propagation: an empirical study," in *Proceedings of the 1988 Connectionist Models Summer School* (D. S. Touretzky, G. E. Hinton, and T. J. Sejnowski, eds.), pp. 38–51, San Francisco, CA: Morgan Kaufmann, 1989.

[17] S. Geisser, "The predictive sample reuse method with applications," *Journal of the American Statistical Association*, vol. 70, pp. 320–328, 1975.

[18] F. J. Smieja, “Neural network constructive algorithms: trading generalization for learning efficiency?,” *Circuits Syst. Signal Process.*, vol. 12, no. 2, pp. 331–374, 1993.

[19] T.-Y. Kwok and D.-Y. Yeung, “Constructive algorithms for structure learning in feedforwardneural networks for regression problems,” *IEEE Transactions on Neural Networks*, vol. 8, pp. 630–645, May 1997.

[20] R. D. Reed and R. J. Marks, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1998.

[21] R. Reed, “Pruning algorithms-a survey,” *IEEE Transactions on Neural Networks*, vol. 4, pp. 740–747, Sept. 1993.

[22] Y. LeCun and Y. Bengio, “Convolutional networks for images, speech, and time series,” in *The handbook of brain theory and neural networks*, (Cambridge, MA, USA), pp. 255–258, MIT Press, 1998.

[23] Y. LeCun, L. Jackel, B. Boser, J. Denker, H. Graf, I. Guyon, D. Henderson, R. Howard, and W. Hubbard, “Handwritten digit recognition: applications of neural network chips and automatic learning,” *Communications Magazine, IEEE*, vol. 27, pp. 41–46, Nov 1989.

[24] K. Oh and K. Jung, “Gpu implementation of neural networks,” *Pattern Recognition*, vol. 37, pp. 1311–1314, June 2004.

[25] C. J. G. Orellana, R. G. Caballero, H. M. G. Velasco, and F. J. L. Aligué, “Neusim: A modular neural networks simulator for beowulf clusters,” in *IWANN '01: Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks*, (London, UK), pp. 72–79, Springer-Verlag, 2001.

[26] E. Schikuta, “Neuroweb: an internet-based neural network simulator,” in *Tools with Artificial Intelligence, 2002. (ICTAI 2002). Proceedings. 14th IEEE International Conference on*, pp. 407–412, 2002.

[27] J. Zhu and P. Sutton, “Fpga implementation of neural networks - a survey of a decade of progress,” in *13th International Conference on Field-Programmable Logic and Applications (FPL 2003)*, 2003.

[28] L. Vapnik, “Estimation of dependencies based on small number of observations,” in *Computational Intelligence for Financial Engineering, 1995.,Proceedings of the IEEE/IAFE 1995*, (New York, NY, USA), Apr. 1995.

[29] B. E. Boser, I. M. Guyon, and V. N. Vapnik, “A training algorithm for optimal margin classifiers,” in *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, (New York, NY, USA), pp. 144–152, ACM, 1992.

[30] R. Fletcher, *Practical methods of optimization; (2nd ed.)*. New York, NY, USA: Wiley-Interscience, 1987.

[31] C. Cortes and V. Vapnik, “Support-vector networks,” *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.

[32] C. J. C. Burges, “A tutorial on support vector machines for pattern recognition,” *Data Min. Knowl. Discov.*, vol. 2, no. 2, pp. 121–167, 1998.

[33] J. C. Platt, “Fast training of support vector machines using sequential minimal optimization,” in *Advances in kernel methods: support vector learning*, (Cambridge, MA, USA), pp. 185–208, MIT Press, 1999.

[34] E. Osuna, R. Freund, and F. Girosi, "An improved training algorithm for support vector machines," in *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, (Amelia Island, FL, USA), pp. 276–285, Sept. 1997.

[35] L. Zanni, T. Serafini, and G. Zanghirati, "Parallel software for training large scale support vector machines on multiprocessor systems," *J. Mach. Learn. Res.*, vol. 7, pp. 1467–1492, 2006.

[36] Y.-H. Dai and R. Fletcher, "New algorithms for singly linearly constrained quadratic programs subject to lower and upper bounds," *Math. Program.*, vol. 106, no. 3, pp. 403–421, 2006.

[37] E. G. Birgin, J. M. Martnez, and M. Raydan, "Nonmonotone spectral projected gradient methods on convex sets," *SIAM J. on Optimization*, vol. 10, no. 4, pp. 1196–1211, 2000.

[38] Y.-H. Dai and R. Fletcher, "Projected barzilai-borwein methods for large-scale box-constrained quadratic programming," *Numer. Math.*, vol. 100, no. 1, pp. 21–47, 2005.

[39] R. Fletcher, "On the barzilai-borwein method," in *Optimization and Control with Applications*, vol. 96 of *Applied Optimization*, pp. 235–256, Springer US, 2005.

[40] T. Joachims, "Making large-scale support vector machine learning practical," in *Advances in kernel methods: support vector learning*, (Cambridge, MA, USA), pp. 169–184, MIT Press, 1999.

[41] C.-J. Lin, "On the convergence of the decomposition method for support vector machines," *IEEE Transactions on Neural Networks*, vol. 12, pp. 1288–1298, Nov. 2001.

[42] H. P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik, "Parallel support vector machines: The cascade svm," in *NIPS*, 2004.

[43] B.-L. Lu, K.-A. Wang, and Y.-M. Wen, "Comparison of parallel and cascade methods for training support vector machines on large-scale problems," in *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, vol. 5, pp. 3056–3061, Aug. 2004.

[44] B.-L. Lu, K.-A. Wang, M. Utiyama, and H. Isahara, "A part-versus-part method for massively parallel training of support vector machines," in *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 1, July 2004.

[45] Y. Wen and B.-L. Lu, "A cascade method for reducing training time and the number of support vectors," in *ISNN (1)*, pp. 480–486, 2004.

[46] IBM Systems and Technology Group, *Cell Broadband Engine Architecture*, 1.02 ed., October 2007. [Online] 24 August 2008.[1]

[47] IBM Systems and Technology Group, *Cell Broadband Engine Registers*, 2.1 ed., March 2007. [Online] 24 August 2008.[1]

[48] IBM, *PowerPC User Instruction Set Architecture Book I*, 2.02 ed., January 2005.

[49] IBM, *PowerPC Virtual Environment Architecture Book II*, 2.02 ed., January 2005.

[50] IBM, *PowerPC Operating Environment Architecture Book III*, 2.02 ed., January 2005.

[51] IBM Systems and Technology Group, *Cell Broadband Engine Programming Handbook*, 1.1 ed., April 2007. [Online] 24 August 2008.[1]

[52] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, “The potential of the cell processor for scientific computing,” in *CF '06: Proceedings of the 3rd conference on Computing frontiers*, (New York, NY, USA), pp. 9–20, ACM, 2006.

[53] S. Olivier, J. Prins, J. Derby, and K. Vu, “Porting the gromacs molecular dynamics code to the cell processor.,” in *IPDPS*, pp. 1–8, IEEE, 2007.

[54] Sony Computer Entertainment Inc. , “Playstation®3 enables folding@home™ to be recognized by guinness world records™ as world's most powerfull distributed computing network,” November 2007. [Online] 24 August 2008. http://www.scei.co.jp/corporate/release/071101e.html

[55] G. Khanna, “Playstation3 gravity grid.” [Online] 24 August 2008. [ONLINE] 24 August 2008. <http://gravity.phy.umassd.edu/ps3.html>

[56] B. Kehaly, “Axion racing playstation3 video processing,” August 2007. [Online] 24 August 2008. <http://www.axionracing.com/Company/PS3_PressRelease.htm>

[57] N. Breese, “Overview of spu-optimized cryptography/"crackstation",” tech. rep., Security-Assessment.com, 2008.

[58] D. P. Scarpazza, O. Villa, and F. Petrini, “Peak-performance dfa-based string matching on the cell processor,” in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, (Long Beach, CA,), pp. 1–8, Mar. 2007.

[59] IBM Systems and Technology Group, “IBM full-system simulator for the cell broadband engine processor.” [Online] 24 August 2008. <http://www.alphaworks.ibm.com/tech/cellsystemsim>

[60] IBM Systems and Technology Group, *Cell Broadband Engine Programming Tutorial*, 2.1 ed., March 2007. [Online] 24 August 2008.[1]

[61] IBM Systems and Technology Group, “IBM assembly visualizer for cell broadband engine.” [Online] 24 August 2008. <http://www.alphaworks.ibm.com/tech/asmvis>

[62] IBM Systems and Technology Group, “Post-link optimization for linux on power.” [Online] 24 August 2008. <http://www.alphaworks.ibm.com/tech/fdprpro>

[63] M. Acton and E. Christensen, “Gdc 2008 - insomniac spu programming,” in *Game Developer's Conference*, 2008.

[64] L. Zanni, “An improved gradient projection-based decomposition technique for support vector machines,” in *Computational Management Science*, vol. 3, pp. 131–145, April 2006.

[65] Systems and Technology Group, “Mathematical acceleration subsystem.”, [Online] 24 August 2008. <http://www-01.ibm.com/software/awdtools/mass>

[66] Systems and Technology Group, “Accuracy information for the mass libraries for the cbe spu.” [Online] 23 August 2008. <http://www-1.ibm.com/support/docview.wss?rs=2021&context=SSVKBV&q1=mass_cbeppu_docs&uid=swg27009549>

[67] Systems and Technology Group, “Performance information for the mass libraries for the cbe spu.” [Online] 23 August 2008. <http://www-1.ibm.com/support/docview.wss?rs=2021&context=SSVKBV&q1=mass_cbeppu_docs&uid=swg27009548>

[68] Y. LeCun and C. Cortes, “The mnist database of handwritten digits.” [Online] 24 August 2008. <http://yann.lecun.com/exdb/mnist/>

[69] M. O'Neill, “Neural network for recognition of handwritten digits,” December 2006. [Online] 24 August 2008. <http://www.codeproject.com/KB/library/NeuralNetRecognition.aspx>

[70] G. Z. Thomas Serafini, Luca Zanni, “Parallel gpdt: Parallel gradient projection-based decomposition technique.” [Online] 24 August 2008. <http://dm.unife.it/gpdt/>

[71] J. A. Blackard, *Comparison of neural networks and discriminant analysis in predicting forest cover types*. PhD thesis, Colorado State University, Fort Collins, CO, USA, 1998. Adviser-Denis J. Dean.

[72] D. N. A. Asuncion, “UCI machine learning repository,” 2007. [Online] 24 August 2008. <http://mlearn.ics.uci.edu/MLRepository.html>

[1]Updated documents are available at the IBM Cell Broadband Engine resource center at <http://www.ibm.com/developerworks/power/cell/>